ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E IMPLEMENTAZIONE DI UNA INCARNAZIONE BIOCHIMICA PER IL SIMULATORE ALCHEMIST

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Co-relatore
Dott.ssa SARA MONTAGNA

Presentata da
GABRIELE GRAFFIETI

Prima Sessione di Laurea
Anno Accademico 2015 – 2016

Remember to look up at the stars
and not down at your feet.
Try to make sense of what you see
and wonder about what makes the universe exist.
Be curious.
And however difficult life may seem,
there is always something you can do and succeed at.
It matters that you don't just give up.
— **Stephen W. Hawking**

A mia madre e mio padre, che in ogni situazione mi saranno sempre accanto.

Indice

Introduzione **ix**

1 Introduzione alla biologia cellulare **1**
1.1 La cellula . . . 2
1.1.1 Principali componenti cellulari . . . 2
1.1.2 Membrana cellulare . . . 5
1.2 Sistemi multicellulari . . . 7
1.2.1 Segnalazione cellulare . . . 7
1.2.2 Giunzioni cellulari . . . 9

2 Introduzione al simulatore Alchemist **13**
2.1 Architettura . . . 13
2.1.1 Reazioni . . . 15
2.2 Motore di simulazione . . . 17
2.2.1 Kinetic Monte Carlo . . . 19
2.2.2 Algoritmo di Gillespie . . . 21
2.2.3 Ottimizzazioni . . . 24
2.2.4 Implementazione . . . 28
2.3 Tool di sviluppo . . . 29
2.3.1 Continuous integration . . . 30

3 Analisi dei requisiti **33**
3.1 Requisiti cellulari . . . 33
3.1.1 Reazioni biochimiche . . . 33
3.1.2 Equilibrio delle reazioni chimiche . . . 34
3.2 Requisiti multi-cellulari . . . 35
3.2.1 Vicinato . . . 35
3.2.2 Reazioni che coinvolgono il vicinato . . . 36
3.2.3 Reazioni che coinvolgono l'ambiente . . . 37
3.3 Requisiti sulle giunzioni . . . 37
3.3.1 Creazione di giunzioni . . . 37
3.3.2 Distruzione di giunzioni . . . 38

3.3.3 Comunicazione attraverso giunzioni 38
3.4 Interfacciamento con il sistema 38
3.4.1 Specifica delle reazioni chimiche 40
3.4.2 Risultati delle simulazioni 41

4 Design 43
4.1 Mappa dei requisiti su Alchemist 43
4.1.1 Molecole 44
4.1.2 Nodi 44
4.1.3 Reazioni 46
4.1.4 Giunzioni 49
4.2 Reazioni con cellule vicine 51
4.2.1 Unione delle condizioni 51
4.2.2 Selezione del vicino 53
4.2.3 Casi particolari 54
4.3 Giunzioni cellulari 56
4.3.1 Creazione 56
4.3.2 Distruzione 56
4.4 Movimento 57
4.4.1 Vicinato con movimento cellulare 57
4.4.2 Problemi 57
4.4.3 Rottura di giunzioni 60
4.5 Linguaggio Biochemistry 62
4.5.1 Costrutti fondamentali 63
4.5.2 Reazioni valide 66
4.5.3 Reazioni non valide 67

5 Implementazione 69
5.1 Linguaggio Biochemistry 69
5.1.1 Strumenti utilizzati 70
5.1.2 Grammatica 71
5.1.3 Visitor 73
5.2 Incarnazione Biochemistry 76
5.2.1 Nodi cellulari 76
5.2.2 Reazioni 77
5.2.3 Reazioni multi-cellulari 78
5.2.4 Giunzioni 80

6 Test 83
6.1 Test intracellulare 84
6.2 Test della comunicazione 86

6.3 Test sulle giunzioni 88
6.4 Test di modelli reali 89
6.5 Valutazione finale 94

Conclusioni **97**

Introduzione

La simulazione di sistemi fisici è da sempre utilizzata dagli scienziati per studiare e comprendere i più disparati fenomeni naturali. Nel corso del *XX* secolo, con l'avvento dell'era informatica, la simulazione attraverso il computer è diventata pervasiva in qualsiasi campo scientifico e tecnologico. I motivi di questo trend sono i più disparati, tra cui il più ovvio è forse l'impossibilità di riprodurre in laboratorio alcuni fenomeni naturali. Si pensi ad esempio alla simulazioni di supernove [1] o di uragani, fenomeni che, se anche fosse possibile riprodurre, sarebbe comunque troppo rischiosa una loro osservazione diretta. Un'altra importante motivazione per l'uso della simulazione al computer è la troppa pericolosità del sistema. È il caso, ad esempio, dei simulatori di volo: è infatti troppo rischioso addestrare aspiranti piloti su veri aerei.

La simulazione, però, non si limita allo studio di fenomeni altrimenti non facilmente osservabili, ma uno dei suoi scopi principali è quello di fornire una predizione futura sull'evoluzione del sistema. Un esempio di questo tipo di simulazione sono le previsioni meteo: esse, attraverso l'uso di complessi modelli matematici e all'osservazione dello stato meteorologico attuale, riescono a produrre una previsione dello stato futuro del sistema, con una precisione molto alta per previsioni di qualche giorno.

Uno dei campi scientifici che ha tratto particolare vantaggio da questo tipo di simulazione computerizzata è la biologia. Conoscendo il modello che governa la vita di una cellula, infatti, è possibile simularne il comportamento e la risposta ad eventuali stimoli esterni. Si è in grado, quindi, di valutare l'efficacia dei farmaci nella lotta contro molte gravi malattie senza dover testarli sui pazienti [2]. La simulazione consente, inoltre, di studiare *in silico* l'evoluzione nel tempo di una patologia, come ad esempio quella di una massa tumorale, potendo cambiarne diversi parametri: in questo modo è possibile simulare l'evolvere della malattia partendo da condizioni iniziali anche molto diverse. I modelli cellulari sono però estremamente complessi da riprodurre, e solamente nel 2012 è stato simulato il completo ciclo vitale di una cellula: il batterio *Mycoplasma genitalium* [3].

Scopo di questo elaborato di tesi è la modellazione e l'implementazione di una estensione del simulatore *Alchemist*, denominata *Biochemistry*, che per-

metta di simulare un ambiente multi-cellulare. Al fine di simulare il maggior numero possibile di processi biologici, il simulatore dovrà consentire di modellare l'eterogeneità cellulare attraverso la modellazione di diversi aspetti dei sistemi cellulari, quali: reazioni intracellulari, segnalazione tra cellule adiacenti, giunzioni cellulari e movimento. Dovrà, inoltre, essere ammissibile anche l'esecuzione di azioni impossibili nel mondo reale, come la distruzione o la creazione dal nulla di molecole chimiche. Questo proprio per definizione di simulazione, che non solo deve rispondere alla domanda "come evolve il sistema", ma anche a quella "cosa succede se eseguo questa azione".

Come detto il motore di simulazione usato è Alchemist. Alchemist è un simulatore open-source e *general-purpose* (ovvero non specifico per un particolare dominio), sviluppato negli ultimi anni all'interno dell'Università di Bologna. Esso è stato scelto per l'utilizzo di molti concetti mutuati dalle scienze chimiche e biologiche, che hanno semplificato l'implementazione di questa sua estensione.

Obiettivo specifico di questa tesi è stato la progettazione e lo sviluppo dei seguenti processi biochimici: reazioni chimiche intracellulari, scambio di molecole tra cellule adiacenti, creazione e distruzione di giunzioni tra le stesse. È stata dunque posta particolare enfasi nella modellazione delle reazioni tra cellule vicine, il cui meccanismo è simile a quello usato nella segnalazione cellulare. Ogni parte del sistema è stata modellata seguendo fenomeni realmente presenti nei sistemi multi-cellulari, e documentati in letteratura. Per la specifica delle reazioni chimiche, date in ingresso alla simulazione, è stata necessaria l'implementazione di un *Domain Specific Language* (DSL) che consente la scrittura di reazioni in modo simile al linguaggio naturale, consentendo l'uso del simulatore anche a persone senza particolari conoscenze di biologia. La semplicità del linguaggio dovrà essere un punto cardine, poiché dovrà permettere l'uso del simulatore a persone con scarse conoscenze informatiche e di programmazione.

La correttezza del progetto è stata validata tramite test compiuti con dati presenti in letteratura e inerenti a processi biologici noti e ampiamente studiati.

La relazione di questo progetto è suddivisa in diverse parti, che ripercorrono le fasi di studio, analisi, progettazione ed implementazione che hanno portato allo sviluppo di una simulazione completa e realistica. La relazione è suddivisa nel seguente modo:

- Nel primo capitolo è presentata una breve introduzione alla biologia della cellula, e alla comunicazione in ambiente multi-cellulare.

- Il secondo capitolo si occupa di descrivere il simulatore Alchemist: in particolare la sua architettura e gli strumenti utilizzati per lo sviluppo. Sono descritti in modo formale anche gli algoritmi usati dal motore di simulazione.

- Il terzo capitolo ha lo scopo di descrivere e analizzare i requisiti del progetto presentato in questa tesi di Laurea.
- Nel quarto capitolo si tratta il design dell'applicazione sviluppata, con particolare enfasi sulle tecniche ingegneristiche adottate.
- Il quinto capitolo descrive l'implementazione del progetto, mentre il sesto discute i test compiuti, e presenta alcune valutazioni sui risultati ottenuti.

Capitolo 1

Introduzione alla biologia cellulare

Questo capitolo ha lo scopo di introdurre alcuni concetti alla base della biologia cellulare. Molti dei comportamenti presentati in questo capitolo saranno poi oggetti di studio nella modellazione ed implementazione dell'incarnazione Biochemistry.

La cellula è la più piccola struttura classificata come essere vivente[1]. Essa è l'unità fondamentale con cui sono composti tutti gli organismi viventi conosciuti. Alcuni di essi sono formati da una sola cellula: è il caso degli organismi unicellulari, come i batteri. La maggior parte degli esseri viventi a noi familiari sono, al contrario, formati da un numero molto elevato di cellule. Negli organismi multicellulari, inoltre, le cellule possono essere anche molto differenziate tra loro. Si pensi ad esempio alla differenza tra cellule epiteliali e cellule nervose.

Le cellule si dividono in due grandi gruppi: *eucariote* e *procariote*. La maggiore differenza tra i due gruppi è data dalla presenza, nelle cellule eucariote, di un nucleo cellulare ben definito e separato dal resto della cellula da una membrana. Solo all'interno del nucleo è contenuto il materiale genetico, sotto forma di DNA, organizzato in cromosomi. Al contrario, nelle cellule procariote, è presente un nucleoide, in cui è concentrato il materiale genetico, non diviso dalla cellula da alcuna barriera. Tutti gli organismi multi-cellulari sono composti da cellule eucariote. La trattazione, da qui in avanti, prenderà in considerazione solo quest'ultimo tipo di cellule.

[1]La classificazione dei virus come esseri viventi è tutt'oggi materia di dibattito [4].

1.1 La cellula

La cellula è separata dall'ambiente esterno da una barriera, detta *membrana citoplasmatica*. All'interno di essa si trovano un insieme di componenti, ognuno con un preciso compito, che concorrono al corretto funzionamento della cellula. Questi componenti sono sospesi in una sostanza semifluida e gelatinosa detta *citosol*. Il numero di componenti (o organelli) contenuto all'interno delle cellule non è costante, al contrario, si differenzia in modo marcato in cellule di tipo diverso. La più profonda distinzione tra cellule eucariote si ha tra cellule animali e cellule vegetali. Le seconde, infatti, possiedono un organello, detto *cloroplasto*, che ha lo scopo di compiere la fotosintesi clorofilliana, impossibile nelle cellule animali. Un'altra differenza sta nella barriera tra cellula ed ambiente esterno. Le cellule animali presentano una barriera semifluida, che può modificare la propria forma e dimensioni. Le cellule vegetali, invece, presentano una *parete cellulare*, formata principalmente da cellulosa, che ha lo scopo di mantenere la forma delle cellule e permettere alla pianta di opporsi alla forza di gravità.

La cellula immagazzina l'energia necessaria allo svolgimento di tutte le sua funzioni in una molecola, detta ATP (*adenosina trifosfato*). Essa, tramite idrolisi, si divide in ADP (*adenosina bifosfato*) e un gruppo fosfato, producendo una notevole quantità di energia, utilizzata dalla cellula per i più svariati compiti, tra i quali:

Lavoro chimico Ovvero realizzazione di reazioni chimiche non possibili senza un apporto esterno di energia.

Lavoro di traporto Ovvero il trasporto di molecole attraverso la membrana in direzione opposta rispetto al gradiente di concentrazione.

Lavoro meccanico Come ad esempio la contrazione di cellule muscolari.

La sintesi dell'ATP avviene continuamente all'interno della cellula, attraverso complesse reazioni molecolari. L'ATP è di solito prodotto partendo dal glucosio, attraverso un processo detto *respirazione cellulare*. Le cellule vegetali, al contrario, lo producono attraverso la fotosintesi clorofilliana.

Le cellule si riproducono attraverso la divisione cellulare, un procedimento che, partendo da una cellula detta *cellula madre*, forma due cellule completamente identiche.

1.1.1 Principali componenti cellulari

Una cellula, proprio come il corpo umano, è formata da un insieme di organelli, ognuno con un compito specifico. Assieme essi rendono possibile

l'attività e la vita della cellula. In questa sezione sono brevemente presentati i principali organelli e descritte le loro funzioni.

Nucleo Il nucleo cellulare è il componente di dimensione maggiore all'interno di una cellula eucariotica. Esso è separato dal citoplasma da una doppia membrana detta *involucro nucleare*. L'involucro è perforato da strutture a forma di poro, che regolano l'ingresso e l'uscita di biomolecole. Il nucleo contiene al suo interno il DNA, organizzato sotto forma di cromosomi. Il nucleo guida la sintesi proteica, sintetizzando l'RNA messaggero (mRNA) in base alle informazioni contenute nel DNA. Successivamente alla sua sintetizzazione l'mRNA viaggia attraverso i pori e viene trasportato nel citoplasma fino ai ribosomi. Essi si occupano di tradurre l'mRNA in proteine.

Ribosomi I ribosomi sono i componenti cellulari deputati alla sintesi proteica. Essi sono maggiormente presenti nelle cellule che presentano una elevata produzione di proteine, come quelle pancreatiche. Essi si distinguono in ribosomi *liberi*, ovvero dispersi nel citoplasma e ribosomi *legati*, che sono associati alla parete esterna del reticolo endoplasmatico. I due tipi di ribosomi sono identici e possono alternarsi nei ruoli. Essi, a partire dall'mRNA prodotto nel nucleo, sintetizzano tutti i diversi tipi di proteine usati dalla cellula.

Reticolo endoplasmatico Il reticolo endoplasmatico (RE) è un complesso insieme di strutture tubolari e cavità, dette cisterne. Nelle cellule si rinvengono due tipi di RE, il RE *rugoso* e il RE *liscio*. Il primo è così chiamato poiché sulla sua superficie sono presenti numerosi ribosomi. Il principale compito del RE rugoso è la secrezione, all'esterno della cellula, delle proteine prodotte dai ribosomi ad esso legati. Un altro importante compito del RE rugoso è la produzione delle membrane destinate alla cellula. Lo scopo del RE liscio è, invece, differente in base al tipo di cellula in cui si trova. Ad esempio nelle cellule epatiche esso rilascia enzimi che partecipano alla detossificazione di sostanze nocive, come alcol o barbiturici.

Apparato del Golgi L'apparato del Golgi può essere immaginato come un centro nel quale i prodotti sintetizzati dai vari organelli della cellula vengono modificati, smistati, immagazzinati ed infine spediti verso la loro destinazione finale. Esso è costituito da un insieme di sacche appiattite delimitate da membrane, dette cisterne. Una cisterna presenta due facce: quella detta *cis* è deputata alla ricezione del materiale, mentre quella opposta, detta *trans*, ha come compito la sua spedizione. Una molecola viene quindi ricevuta dalla faccia *cis*, attraversa l'apparato del Golgi e

viene spedita dalla faccia *trans*. Durante il tragitto tra le due facce essa può subire varie trasformazioni e può essere unita ad altre molecole.

Lisosomi Un lisosoma è una vescicola contenente enzimi digestivi, impiegati dalle cellule animali per la digestione di macromolecole. Questi enzimi sono prodotti dal RE rugoso ed elaborati nell'apparato del Golgi, dove vengono formati i lisosomi. Molti organismi unicellulari praticano *fagocitosi* per ricavare i nutrienti essenziali alla loro sopravvivenza. Un organismo viene "circondato" dalla cellula e inglobato in essa, all'interno di un *vacuolo alimentare*. Esso si unisce poi a uno o più lisosomi, che digeriscono l'organismo fagocitato, sintetizzandone molecole energetiche poi disperse nel citoplasma. Anche i globuli bianchi umani praticano fagocitosi, inglobando batteri e agenti patogeni dannosi. Nel caso un organello della cellula sia danneggiato, esso viene auto-digerito dai lisosomi, riciclando materiali utili alla sintesi di nuovi organelli.

Mitocondri I mitocondri sono i principali organelli deputati alla produzione di ATP. Essi sono presenti in numero maggiore nelle cellule che presentano una attività metabolica intensa, come quelle muscolari. Essi sono formati da una doppia membrana, in cui quella più interna è ripiegata in modo da formare creste e ripiegamenti, che ne aumentano la superficie. I mitocondri contengono al loro interno enzimi, ribosomi e DNA mitocondriale, indispensabili per la respirazione cellulare.

Citoscheletro La funzione più evidente svolta dal citoscheletro è quella di fornire un supporto meccanico alla cellula, conferendole una specifica forma che può essere mantenuta nel tempo. Esso fornisce un punto di ancoraggio ai vari organelli, ma, a differenza dello scheletro degli animali, esso può cambiare rapidamente conformazione, dando alla cellula una forma diversa. Il citoscheletro è formato da un insieme di filamenti, di diametro diverso, ancorati tra loro. Questi micro-filamenti sono usati come binari all'interno della cellula. Una vescicola che si sposta dal reticolo endoplasmatico all'apparato del Golgi, infatti, segue i filamenti interni alla cellula per arrivare a destinazione. Il citoscheletro è, inoltre, legato alla *matrice extra-cellulare*, un insieme di tuboli che tiene unite cellule vicine. Tramite questo collegamento è possibile anche il movimento della cellula nell'ambiente circostante.

Un'illustrazione della cellula con tutti gli organelli è visibile in Figura 1.1

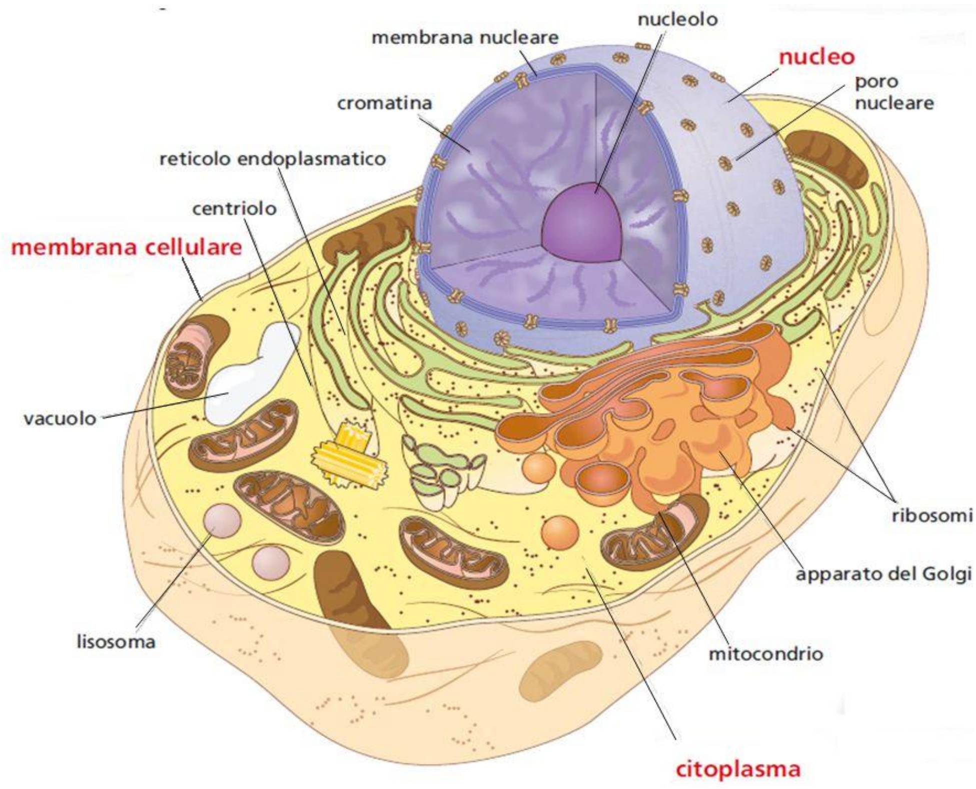


Figura 1.1: Una cellula animale.

1.1.2 Membrana cellulare

Come detto, la cellula è separata dall'ambiente esterno tramite una membrana detta *membrana citoplasmatica*. Essa è formata da un insieme di molecole, dette *fosfolipidi*. Essi presentano una struttura particolare: hanno infatti una parte della molecola idrofila (che forma legami con le molecole di acqua) ed una parte idrofoba (che, al contrario, non ne forma). Un fosfolipide è visibile in Figura 1.2. La membrana è, quindi, formata da un doppio strato di fosfolipidi, le cui teste idrofile sono rivolte verso l'esterno della membrana, mentre le code, idrofobe, sono rivolte verso l'interno.

La membrana cellulare non è una lamina statica costituita da molecole saldamente legate tra loro, ma una barriera fluida che può muoversi e riposizionarsi in base alla forma della cellula. La fluidità della membrana è, infatti, simile a quella dell'olio d'oliva.

La membrana non è però unicamente formata da fosfolipidi. Immerse in essa sono presenti numerosissime proteine, dette *proteine di membrana*, responsabili della maggior parte delle funzioni svolte dalla membrana stessa. Tra queste funzioni si annoverano: il trasporto di molecole dalla cellula all'am-

biente esterno, e viceversa, la trasduzione dei segnali extra-cellulari provenienti da cellule vicine, il riconoscimento tra cellule, l'adesione intracellulare e alla matrice extracellulare.

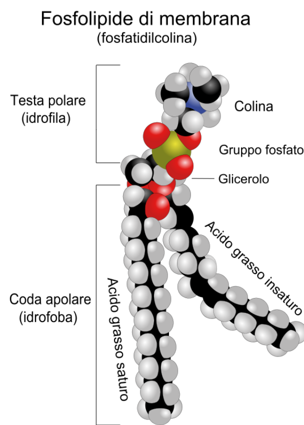


Figura 1.2: Un fosfolipide. Sono visibili la testa idrofila e le code idrofobe. (fonte Wikipedia)

La membrana cellulare non è totalmente impermeabile. Essa consente il passaggio di numerose molecole, che dal mezzo extra-cellulare migrano verso l'interno della cellula, per poi essere usate nelle varie reazioni chimiche. Per la sua natura fosfolipidica, la membrana citoplasmatica consente il passaggio attraverso essa di molecole apolari, come gli idrocarburi, il biossido di carbonio o l'ossigeno. Le molecole polari, come l'acqua o il glucosio, attraversano il doppio strato fosfolipidico con notevole lentezza; gli ioni, addirittura, data la loro polarità, non possono attraversare la membrana. Alcune delle proteine ancorate alla membrana la attraversano da parte a parte, formando al loro interno un tunnel che attraversa la membrana. Esse consentono il passaggio alle molecole che non riuscirebbero ad attraversare facilmente la membrana, come l'acqua. Altre proteine non formano semplicemente un passaggio per le molecole, ma cambiano la loro conformazione per favorire in modo più netto il passaggio. Proprio come le chiuse usate nei canali di navigazione esse presentano un lato aperto ed uno chiuso. Quando una molecola entra dentro la proteina essa cambia forma, chiudendo il lato aperto e aprendo quello chiuso. In questo modo la molecola viene trasportata dall'esterno all'interno della cellula o viceversa. Tutti i tipi di trasporto fin qui elencati sono detti *passivi*, poiché non usano energia per lo spostamento di molecole. Esse, infatti, si spostano in base al loro gradiente di concentrazione, andando quindi dalle zone in cui la concentrazione è maggiore a quelle in cui è minore.

Trasporto attivo

Il trasporto passivo, visto in precedenza, non è sufficiente per mantenere un ambiente cellulare sano. Le molecole, infatti, si muovono soltanto attraverso il loro gradiente di concentrazione, e ciò non è sempre il comportamento voluto. Il trasporto attivo consente di trasportare molecole in modo contrario al loro gradiente di concentrazione, ovvero da dove la concentrazione è più bassa a dove è più alta. Al contrario del trasporto passivo, che avviene senza dispendio energetico, il trasporto attivo consuma energia. Tutte le proteine coinvolte nel

trasporto attivo prendono il nome di *trasportatori*, e consentono alla cellula di mantenere al suo interno una concentrazione di molecole diversa da quella presente all'esterno. Ad esempio una cellula animale possiede al suo interno una concentrazione maggiore di ioni potassio e una minore di ioni sodio rispetto all'ambiente extracellulare. Un esempio di trasporto attivo è dato dalla pompa sodio-potassio, la quale elimina ioni sodio dalla cellula e trasporta ioni potassio all'interno di essa. Per ogni ciclo essa consuma l'energia data da una singola molecola di ATP.

1.2 Sistemi multicellulari

Tutti gli organismi multi-cellulari sono formati da cellule di diverso tipo che comunicano tra loro. Anche alcuni organismi unicellulari, come i batteri, si uniscono in colonie formate da milioni di cellule, che comunicano ed interagiscono. Si pensa che i primi organismi multi-cellulari, siano nati proprio da colonie di organismi unicellulari.

La comunicazione, o segnalazione, cellulare è indispensabile nel coordinamento delle funzioni di un gruppo di cellule, basti pensare ai segnali inviati dal cervello che consentono di muovere i muscoli delle gambe durante una camminata.

1.2.1 Segnalazione cellulare

La segnalazione cellulare si divide principalmente in due rami: la segnalazione locale e la segnalazione a distanza. La segnalazione locale può avvenire in diversi modi, il più semplice è forse il contatto diretto tra cellule. Esse presentano delle proteine di membrana il cui scopo è proprio quello di legarsi con le corrispondenti proteine di membrana di cellule vicine. Questo processo è detto *riconoscimento cellulare*, ed ha una particolare rilevanza in fenomeni come lo sviluppo embrionale o la risposta immunitaria. In molti casi le molecole che fungono da messaggeri vengono rilasciate dalla cellula nell'ambiente extracellulare. Alcune di esse agiscono solo a breve distanza: è il caso dei *regolatori locali*, i quali possono influenzare solo cellule a vicine alla sorgente del segnale. Essi sono usati dai mammiferi per trasportare i *fattori di crescita*, sostanze che stimolano l'accrescimento e la moltiplicazione delle cellule adiacenti alla sorgente del segnale. Più di una cellula può ricevere e rispondere a questo tipo di segnalazioni, innescando reazioni a catena che diffondono il segnale. Un'altro tipo di segnalazione locale è la *segnalazione sinaptica*. Essa avviene quando un segnale elettrico che attraversa una cellula nervosa è arrivato alla fine della stessa. Esso induce il rilascio di diverse molecole chimiche

che vanno a stimolare la cellula nervosa vicina che, a sua volta, trasdurrà il segnale chimico in un segnale elettrico.

La segnalazione a distanza non avviene tra cellule nelle immediate vicinanze, ma tra parti dell'organismo anche molto distanti tra loro. Nella segnalazione ormonale (anche detta *segnalazione endocrina*), alcune cellule specializzate rilasciano ormoni nel flusso sanguigno, attraverso cui raggiungono cellule bersaglio localizzate in altre parti dell'organismo. Un altro esempio di segnalazione a distanza è dato dagli stimoli nervosi. Essi viaggiano sotto forma di segnali elettrici all'interno delle cellule nervose. Giunti alla fine di una di esse vengono tradotti in segnali chimici che, a loro volta, vengono recepiti dalla cellula nervosa successiva. L'insieme di tutti questi segnali consente la comunicazione tra il cervello e i muscoli presenti in ogni parte del corpo.

La segnalazione cellulare avviene attraverso tre fasi: ricezione, trasduzione e risposta.

Ricezione del segnale

La ricezione rappresenta la capacità, da parte di una cellula bersaglio, di rilevare un segnale proveniente dall'ambiente extracellulare. La ricezione di molecole di segnalazione si compie attraverso specifici recettori di membrana, presenti nelle cellule che "devono ascoltare" un certo segnale. Notare come molte cellule possono ricevere una segnalazione, sia da parte di cellule vicine, sia a distanza, ma solo quelle con il recettore per quel particolare tipo di segnale sono in grado di riceverlo. Una recettore mostra una complementarietà strutturale con le molecole che trasportano il segnale, come una chiave e la rispettiva serratura. Quando un recettore di membrana si unisce ad una molecola segnale esso cambia forma. Ciò è sufficiente ad attivare il recettore, e segnalare all'interno della cellula che è stata ricevuta una segnalazione. Esistono recettori anche all'interno del citoplasma. Essi possono fungere da recettori intermedi (nel caso che il recettore di membrana rilasci una molecola quando attivato, essi si occupano di recepire quella molecola), o da recettori finali, nel caso la molecola bersaglio possa attraversare la membrana citoplasmatica. È il caso, ad esempio, del testosterone, che, trasportato dal flusso sanguigno, raggiunge tutte le cellule del corpo; soltanto quelle con i recettori, però, possono ricevere la segnalazione e attivare la conseguente risposta.

Trasduzione del segnale

In questa fase il segnale cellulare viene trasformato, in modo da poter attivare la risposta cellulare. Essa non avviene in un solo passaggio, ma, in genere, in più tappe distinte. Un vantaggio della divisione in diverse fasi è dato dalla possibilità di amplificare sensibilmente il segnale. Se ad ogni

tappa il numero di molecole che ricevono il segnale è maggiore di quelle che lo trasmettono, il risultato sarà un enorme numero di molecole attive al termine della trasduzione. Inoltre è molto più semplice, per le cellule, coordinare e regolare processi a più fasi rispetto a processi mono-fase.

Quando il segnale arriva ad un recettore, in genere, esso cambia forma, o rilascia nel citoplasma alcune molecole. Questa è la fase uno della trasduzione. Ogni fase userà come segnali di attivazione i prodotti della fase precedente e rilascerà nella cellula diverse molecole, dette prodotti. L'insieme di tutte le fasi porta alla trasformazione del segnale, e al suo movimento dalla membrana alla specifica sede della risposta. Questo processo viene detto *cascata di segnalazione*, e può comprendere anche recettori intermedi.

Risposta al segnale

Alla fine della fase di trasduzione di un segnale si ha sempre un evento di regolazione di una o più attività cellulari. Esso consiste tipicamente nella regolazione della sintesi di un certo tipo di proteine, attivandone o disattivandone la produzione. Un altro tipo di risposta può regolare l'attività di una data proteina, ad esempio, nel caso di proteine canale, esso può venire chiuso o aperto in base a segnali extra-cellulari. Alcuni tipi di segnalazione vengono ritrasmessi dalle cellule che li hanno ricevuti. Questo comportamento consente una maggiore estensione della segnalazione locale.

Una risposta deve, in molti casi, essere attivata da più di una segnalazione. In altri casi, al contrario, un singolo segnale può produrre più risposte all'interno della cellula.

Una volta ricevuto il segnale e prodotta la relativa risposta la cellula deve provvedere alla terminazione di tutte le fasi di ricezione e trasduzione del segnale. Questo è necessario per poter consentire alla cellula di ricevere altri segnali. La molecola di segnalazione si deve quindi staccare dal recettore, che ritorna allo stato ricettivo, e tutte le molecole coinvolte nelle varie fasi di trasduzione devono ritornare inattive, in attesa di un nuovo segnale. Non terminare correttamente una segnalazione può portare a gravi conseguenze, poiché essa sarebbe considerata come continuamente in atto, producendo in modo incontrollato la risposta specifica.

1.2.2 Giunzioni cellulari

Le cellule di organismi multi-cellulari sono, tipicamente, organizzate in tessuti. All'interno di un tessuto le cellule aderiscono, interagiscono e comunicano tra loro attraverso un contatto fisico diretto. Diversamente da quello che si può pensare le membrane cellulari non isolano in modo assoluto la cellula da

quelle vicine. Possono, infatti, formarsi giunzioni cellulari tra le membrane citoplasmatiche di cellule adiacenti, in modo da ancorarle tra loro e di favorire la segnalazione.

Una giunzione è quindi un forte legame instaurato tra due cellule, o una cellula e la matrice extra-cellulare. Le giunzioni sono formate da un insieme di molecole presenti nelle membrane delle due cellule partecipanti. Se esse sono abbastanza vicine, queste molecole instaurano legami chimici tra loro, in grado di connettere le cellule. La creazione di una giunzione può portare ad una risposta interna alla cellula, nello stesso modo della segnalazione cellulare trattata nella sezione precedente. La creazione di una giunzione, quindi, non si limita a collegare le due cellule, ma funge anche da evento segnalatore. La creazione di una giunzione, a volte, necessita anche di molecole presenti nel mezzo extra-cellulare. È il caso delle giunzioni formate da caderina, che necessitano di calcio per legare altre cellule.

Una cellula può avere nello stesso momento un numero molto alto di giunzioni, anche con cellule diverse. Quando una giunzione cellulare viene spezzata le molecole presenti nelle membrane delle cellule rimangono ognuna ancorata alla cellula di origine, in modo da poter formare altre giunzioni.

Le giunzioni si dividono principalmente in tre tipi diversi, qui di seguito esposti:

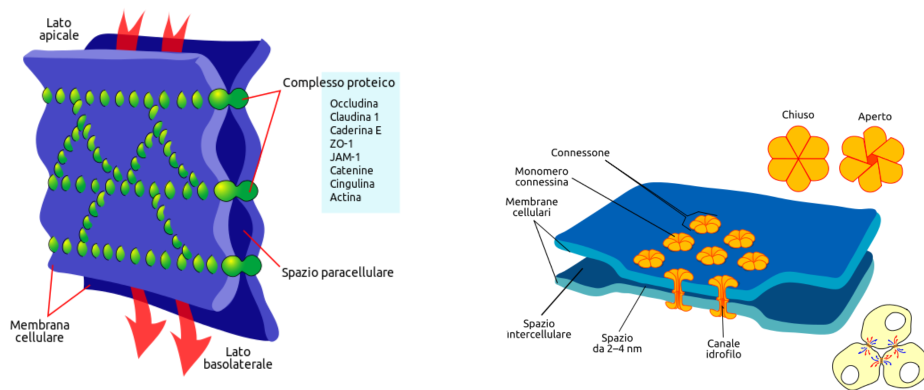
Giunzioni occludenti (*tight junctions*) svolgono una funzione occludente, ovvero, in corrispondenza di questo tipo di giunzioni, le membrane citoplasmatiche delle due cellule risultano unite, per mezzo di specifiche proteine. La stretta adesione cellulare non permette il passaggio di molecole, saldando in modo totale lo spazio tra le due cellule. Esse sono presenti sopratutto nei tessuti epiteliali, come la pelle, impedendo la perdita di materiale attraverso i tessuti.

Giunzioni di ancoraggio (o desmosomi) sono connessioni tra cellule o tra cellula e matrice extra-cellulare. Esse si legano, nel lato cellulare, al citoscheletro, mentre, nel lato extra-cellulare, ai filamenti di cheratina della matrice extracellulare. Esse determinano una stretta adesione tra le cellule (ma non al livello delle giunzioni occludenti). Il loro scopo principale è, quindi, quello di tenere assieme le diverse cellule che formano un tessuto.

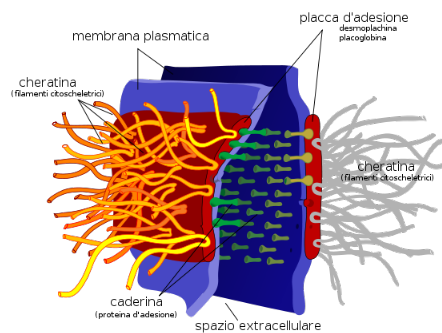
Giunzioni comunicanti (gap junctions) formano canali di comunicazione trans-membrana tra cellule adiacenti. Esse si compongono di proteine di membrana che circondano un poro attraverso cui transitano molecole tra le due cellule. Sono quindi “tunnel” tra i citosol di cellule vicine. Esse sono indispensabili per la comunicazione in diversi tipi di tessuto, come

quello cardiaco. La permeabilità delle *gap junctions* è regolabile dalle cellule: determinati eventi intracellulari possono aumentare o diminuire il numero di molecole che attraversano la giunzione, fino a chiuderla completamente. Questo tipo di giunzioni è presente anche in gruppi di cellule che hanno interesse a mantenere la concentrazione di certe molecole allo stesso livello. Questo è utile sopratutto per regolare in modo coordinato la risposta a certi stimoli.

I diversi tipi di giunzioni presentati sono visibili in Figura 1.3.



(a) Giunzione occludente (b) Giunzione comunicante



(c) Giunzione di ancoraggio

Figura 1.3: Diversi tipi di giunzioni. (fonte Wikipedia)

Capitolo 2

Introduzione al simulatore Alchemist

Il motore di simulazione usato è Alchemist, un simulatore scritto in Java sviluppato come progetto all'interno dell'università di Bologna. La maggior parte dei software di simulazione è *domain specific*, ovvero, al loro interno, può essere modellato soltanto un preciso sistema fisico. Alcuni simulatori, addirittura, consentono di simulare soltanto particolari aspetti di un certo fenomeno. Alchemist, al contrario, si caratterizza per essere *general purpose*, ovvero in grado di simulare una vasta gamma di sistemi. Alchemist è un simulatore *stocastico*, ovvero l'evoluzione del sistema simulato avviene tramite una distribuzione di probabilità. Ciò significa che l'evoluzione del sistema non è deterministica.

Il codice sorgente di Alchemist è inoltre *open source*: ciò consente a chiunque di poter scriverne un'estensione o correggerne eventuali difetti. Un'estensione del simulatore che permetta allo stesso di simulare un dato fenomeno viene detta *incarnazione*. Essa non è altro che l'implementazione dei concetti astratti utilizzati da Alchemist in contesti *domain specific*.

2.1 Architettura

Alchemist è un simulatore costruito in modo modulare. Ogni modulo ha un preciso compito e può servirsi di altri moduli per il suo funzionamento, come mostrato in Figura 2.1. I due moduli sicuramente più importanti sono quello relativo al motore di simulazione (che verrà trattato in una sezione successiva) e il modulo specifico per un'incarnazione. Esso è quello modellato e sviluppato in questo progetto.

Come accennato nell'introduzione, Alchemist si basa su concetti propri delle scienze chimiche, per cui è risultato particolarmente agevole adattarlo

ad ospitare un sistema multi-cellulare. Le principali astrazioni sono, infatti: un *ambiente*, al cui interno è contenuto un insieme di *nodi*. Ogni nodo può contenere al suo interno un insieme di *molecole* con il rispettivo valore di concentrazione. I nodi possono inoltre contenere una o più *reazioni*, a loro volta formate da *condizioni*, *azioni* e una distribuzione probabilistica che ne rappresenta la probabilità di esecuzione. A differenza di molti simulatori chimici o biologici, che modellano un solo compartimento dove avvengono reazioni, Alchemist modella un insieme di compartimenti, ognuno con le proprie molecole e reazioni. I compartimenti, inoltre, possono essere collegato attraverso una *linking rule*. Due compartimenti collegati vengono detti *vicini*. Due nodi vicini possono scambiarsi molecole tra loro, ponendo le basi per la modellazione di un sistema multi-compartimento. Le relazioni di vicinato non sono però fisse nel tempo. I nodi, infatti, possono muoversi, venire eliminati e creati, rompendo o instaurando nuovi collegamenti. Uno schema delle principali astrazioni di Alchemist è visibile in Figura 2.2.

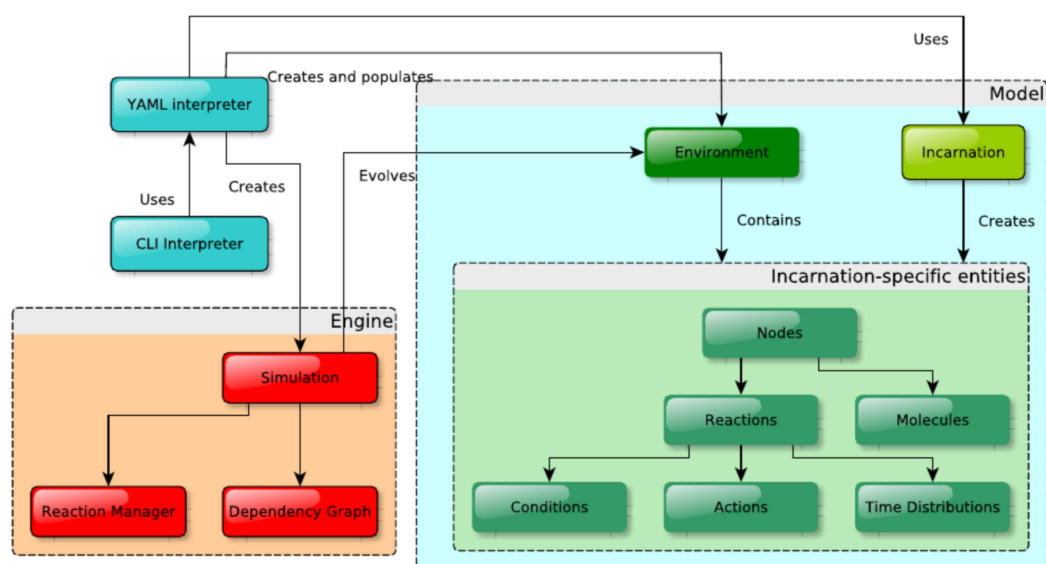


Figura 2.1: In questa immagine sono sintetizzati i principali moduli del simulatore Alchemist. I due interpreti, della console e del file YAML, hanno il compito di creare e popolare una simulazione. Il modulo del motore di simulazione (*engine*) contiene al suo interno gli algoritmi e le strutture dati idonee ad eseguire una simulazione stocastica basandosi sui metodi esposti nella sezione 2.2. Esso evolve l'ambiente che a sua volta contiene le astrazioni specifiche per ogni incarnazione. (fonte [5])

Tutti questi concetti sono completamente generici, e solamente nella fase implementativa di una specifica incarnazione saranno mappati su concetti pro-

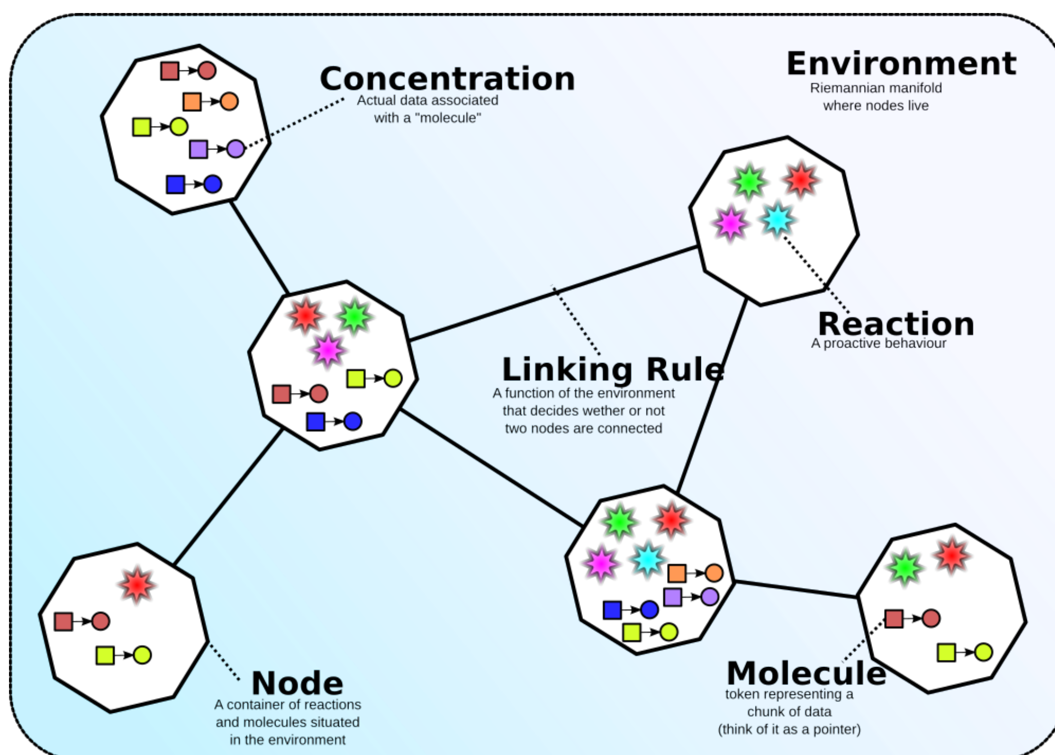


Figura 2.2: Schema che mostra le principali astrazioni di Alchemist. All'interno dell'ambiente sono contenuti diversi nodi, collegati tra loro attraverso una *linking rule*. Ogni nodo al suo interno può contenere molecole, con la rispettiva concentrazione, e reazioni. Questo tipo di modello rappresenta l'astrazione di un sistema multi-compartimento con collegamenti tra i diversi nodi. Esso può essere usato come astrazione per i più disparati sistemi fisici. (fonte [5])

pri del dominio modellato. Ad esempio una molecole può rappresentare una molecola chimica, oppure un programma in esecuzione su un elaboratore. Allo stesso modo un nodo può essere una cellula, ma anche un computer connesso ad una rete di calcolatori.

2.1.1 Reazioni

Uno dei concetti più importanti di quelli finora espressi è quello di reazione. In Alchemist una reazione è formata da tre astrazioni fondamentali:

- Un insieme (anche vuoto) di condizioni. Una condizione è una funzione che, preso in ingresso lo stato corrente della simulazione, restituisce un valore booleano, vero se la condizione è soddisfatta o falso in caso contrario. Se anche una sola condizione non è soddisfatta la reazione non può

aver luogo. Una condizione restituisce anche un numero $p \in \mathbb{R}_{\geq 0}$ detto *propensity*, il quale rappresenta l'influenza della condizione nel calcolo della velocità della reazione (quanto è probabile che la reazione avvenga nel prossimo intervallo temporale). L'uso di questo numero dipende dal tipo di reazione e dalla sua distribuzione temporale.

- Un insieme (anche vuoto) di azioni. Un'azione modella un qualunque cambiamento nell'ambiente di simulazione. Formalmente un'azione è una funzione che fa passare la simulazione dallo stato corrente $\mathbf{x_0}$ allo stato $\mathbf{x}$. Esempi di azioni posso essere il cambiamento della concentrazione di una molecola in un nodo, o il movimento di un nodo all'interno dell'ambiente.

- Una distribuzione temporale, che abbinata ad un numero $r \in \mathbb{R}_{\geq 0}$ detto *rate statico* della reazione consente di ottenere una distribuzione di probabilità che indica quando sarà più probabile che la reazione venga eseguita. Il rate, formalmente, rappresenta il numero di volte che la reazione deve essere eseguita (se tutte le sue condizioni sono valide) per unità di tempo. In base al tipo di reazione la propensity delle condizioni potrebbe essere presa in considerazione nel calcolo del rate. In questo caso il rate viene detto *rate istantaneo*.

Uno schema della composizione di una reazione è visibile in Figura 2.3.

Una simulazione in Alchemist è attiva finché all'interno dell'ambiente vi è almeno una reazione che è possibile eseguire (tutte le condizioni sono valide). Nel momento che nessuna reazione può essere schedulata per l'esecuzione la simulazione volge al termine. Questo perché il sistema può essere evoluto solo grazie alle reazioni. Se nessuna di esse può essere effettuata non avrebbe senso continuare a usare risorse di calcolo per la simulazione di un sistema che non avrà nessuna evoluzione. Le reazioni, dunque, rimangono perennemente all'interno dei nodi, le loro condizioni vengono verificate e, se tutte valide, le azioni associate alla reazione verranno eseguite in un tempo inversamente proporzionale alla grandezza del rate.

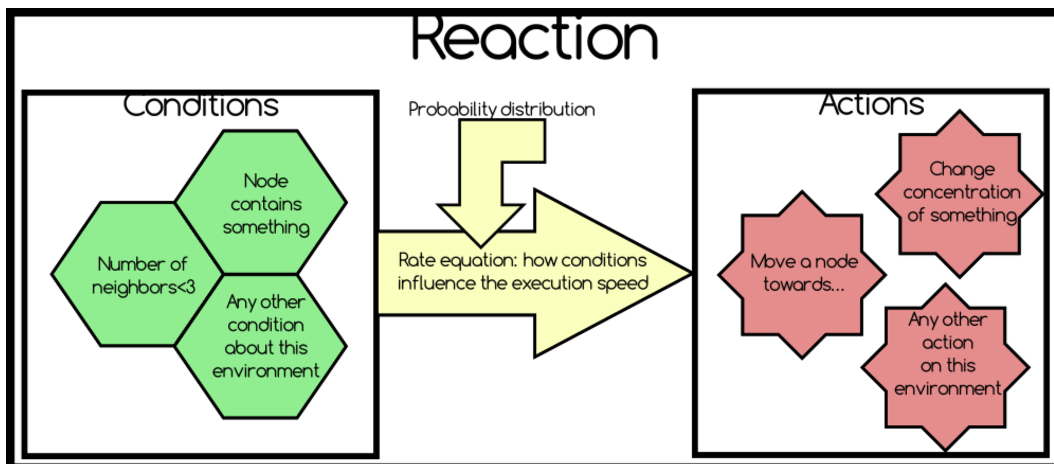


Figura 2.3: Immagine che mostra la composizione di una reazione in Alchemist. Essa è formata da un insieme di condizioni e uno di azioni. Se le prime sono valide la loro propensity potrebbe essere usata, in combinazione con il rate statico della reazione, per il calcolo del rate istantaneo, attraverso una *rate equation*. Questo valore, combinato con una distribuzione probabilistica, rappresenta la probabilità che la reazione venga eseguita in futuro. Se anche una sola condizione non dovesse essere valida il rate sarebbe 0 e la reazione non potrebbe essere effettuata. (fonte [5])

2.2 Motore di simulazione

Esistono molti approcci diversi al problema di simulare un sistema all'elaboratore, e tutti dipendono da come è stato implementato il modello del fenomeno da simulare. Le due principali classi in cui possono essere divisi i simulatori sono:

Continui Il fenomeno è modellato tipicamente attraverso equazioni differenziali, che vengono risolte con metodi numerici. Periodicamente il sistema risolve le equazioni e, tramite il risultato trovato, cambio lo stato interno e l'output. Questi risultati saranno usati come input per il calcolo della prossima iterazione della simulazione. In questo caso il tempo è trattato come una variabile continua.

Discreti Il fenomeno viene modellato attraverso una sequenza discreta di eventi, che accadono ad un certo istante di tempo. Ogni evento, una volta eseguito, attua un cambiamento nel sistema, che passa quindi dallo stato $\mathbf{x_i}$ allo stato $\mathbf{x_{i+1}}$. Questo modello è attualmente il più utilizzato nei simulatori commerciali ed è anche quello implementato da Alchemist.

Ci sono due modi di costruire un simulatore discreto. Il primo, e forse più ovvio, è quello di discretizzare il tempo. Si parta dal tempo t_0 e si scelga un intervallo di tempo $\tau > 0$. Se nell'intervallo temporale $t_0 + \tau$ (*tick*) ci sono eventi essi vengono eseguiti e lo stato del sistema viene conseguentemente cambiato. Il tempo viene inoltre aggiornato al valore $t \leftarrow t + \tau$. Se due eventi separati avvengono nello stesso tick essi sono considerati simultanei. Questo modello prende il nome di *Time Driven*. È facile notare come la grandezza di τ possa influenzare la velocità e la precisione della simulazione. Prendendo un τ troppo piccolo, infatti, il tempo viene continuamente avanzato senza che nessun evento abbia luogo. Ciò può portare ad uno spreco di risorse di calcolo e di tempo impiegato, per completare la simulazione. Allo stesso modo usare un valore di τ troppo grande può portare ad avere molti eventi nello stesso tick, e quindi un'evoluzione del sistema non realistica. Questo modello, inoltre, è inadeguato per sistemi che presentano una distribuzione degli eventi non uniforme. In questo ultimo caso, infatti, la scelta della grandezza del tick può essere solamente un compromesso tra prestazioni e correttezza del risultato.

Un approccio migliore è il cosiddetto *Discrete Event Simulation* (DES). In questo modello di simulazione gli eventi sono simulati uno ad uno, seguendo il loro ordine di schedulazione. Tra due eventi consecutivi si suppone che non siano effettuati cambiamenti nel sistema (questo perché soltanto un evento può aggiornarne lo stato) e quindi il tempo può essere direttamente portato a quello del prossimo evento da eseguire. Se due eventi sono stati schedulati nello stesso istante temporale uno dei due è eseguito per primo, e lo stato cambiato prima che l'altro possa eseguire. Ciò significa che l'ordine in cui due eventi simultanei sono eseguiti può influenzare il resto della simulazione.

Alchemist al suo interno utilizza un DES come modello di simulazione, considerando gli eventi ordinati e avanzando il tempo tra due eventi consecutivi.

Alchemist, come accennato precedentemente, è un simulatore *stocastico*. Ciò significa che l'evoluzione del sistema avviene in modo casuale, attraverso l'uso di numeri random, distribuiti secondo una nota funzione di probabilità. Far evolvere un sistema basandosi su variabili casuali potrebbe portare a risultati non precisi o addirittura totalmente errati. Una soluzione a questo problema è il metodo di *Monte Carlo*. Esso calcola una serie di realizzazioni possibili del problema in esame, cercando di esplorare tutto lo spazio dei parametri del fenomeno. Una volta calcolato un possibile stato del sistema esso compie delle misure. Facendo la media di tutte le misure fatte in tutti i diversi stati possibili si ottiene il valore che il sistema avrà alla fine della computazione.

2.2.1 Kinetic Monte Carlo

Il Kinetic Monte Carlo è un metodo, che si basa sul classico metodo Monte Carlo, usato per simulare l'evoluzione temporale di sistemi fisici. Un fenomeno è modellato attraverso un insieme di stati. Da uno stato è possibile passare ad un insieme di altri stati futuri. Ad ognuna di queste transizioni è associato un numero, detto *rate*, che rappresenta la probabilità che dallo stato corrente si passi allo stato j. Tanto più il rate è alto tanto più è probabile che il sistema evolva dallo stato corrente a quello con rate maggiore. I rate delle diverse transizioni devono essere noti a priori. L'algoritmo, infatti, ha il compito di evolvere il sistema, non di predire la probabilità che lo stesso passi da uno stato ad un'altro.

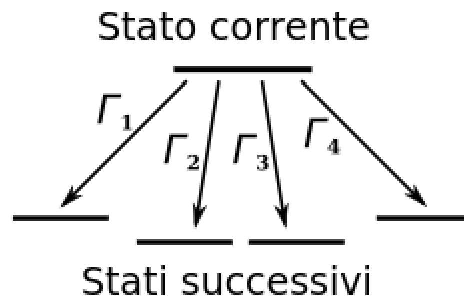


Figura 2.4: Dallo stato corrente è possibile passare ad una moltitudine di altri stati. Ogni transizione ha associato un rate che ne caratterizza la probabilità. (fonte Wikipedia)

Con il sistema nello stato $\mathbf{x_0}$ e al tempo t_0 si hanno quindi i rate associati a tutte le transizioni (r_{kj}) con k stato attuale e j stato successivo. Si computi quindi la loro somma R. La probabilità che una transizione sia eseguita è quindi r_j/R. Tramite l'estrazione di un numero casuale u compreso tra 0 e R si riesce a trovare l'indice j del prossimo stato. Aggiorno quindi $\mathbf{x} \leftarrow \mathbf{x_j}$. Estraggo ora un'altro numero casuale $u' \in (0, 1]$ e aggiorno il tempo come $t \leftarrow t_0 + \Delta T$ con $\Delta T = R^{-1} \ln{(1/u')}$. La relazione tra rate della transazione e probabilità che sia scelta è mostrata in Figura 2.5.

Figura 2.5: Scelta della transizione da eseguire in base al rate. Viene lanciato un numero random tra 0 e r_{tot} e in base all'intervallo in cui esso cade viene scelto lo stato j. Notare come l'ampiezza di un intervallo associato ad uno stato sia proporzionale al corrispettivo rate. (fonte Wikipedia)

In modo più formale il metodo kinetic Monte Carlo funziona nel seguente modo:

1. Si pone il tempo $t \leftarrow t_0$ e lo stato $\mathbf{x} \leftarrow \mathbf{x_k}$.

2. Si forma la lista $\boldsymbol{N}_k$ delle possibili transazioni dallo stato $\mathbf{x_k}$ allo stato $\mathbf{x_j}$. Per ognuna di esse conosco il rate r_{kj}. Se uno stato non è raggiungibile da quello corrente il suo rate r_{kj} sarà uguale a 0.

3. Viene calcolata la funzione cumulativa $\mathbf{R}_{ki} = \sum_{j=1}^{i} r_{kj}$ per i $= 1, \ldots, \boldsymbol{N}_k$.
   La somma di tutti i rate è data da $\boldsymbol{Q}_k = \boldsymbol{R}_{k,N_k}$.

4. Viene estratto un numero random $u \in (0, 1]$.

5. Si trova l'indice i del prossimo stato come $\boldsymbol{R}_{k,i-1} < u\boldsymbol{Q}_k \leq \boldsymbol{R}_{ki}$. Ciò può essere ottenuto efficientemente ordinando la lista dei possibili stati futuri e usare una ricerca binaria per trovare lo stato obiettivo.

6. Viene aggiornato lo stato corrente $\mathbf{x_k} \rightarrow \mathbf{x_i}$

7. Viene estratto un nuovo numero random $u' \in (0, 1]$

8. Il tempo della simulazione viene aggiornato $t = t_0 + \frac{1}{\boldsymbol{Q}_k} \ln\left(\frac{1}{u'}\right)$
   Essendo $\ln(1/u')$ mediamente uguale a 1, il tempo viene aumentato in modo inversamente proporzionale alla somma di tutti i rate. Ciò è sensato poiché se una transizione tra due stati è molto probabile essa verrà eseguita in un intervallo temporale vicino al tempo attuale.

9. Se si è raggiunto $t = t_{max}$ STOP. Altrimenti si ritorna allo step 2.

Si è dimostrato che se i rate associati alle transizioni di stato sono corretti l'algoritmo converge verso un'evoluzione temporale del sistema corretta. Essendo un algoritmo basato sul metodo Monte Carlo, dovrà essere reiterato un numero ragionevole di volte per poter coprire tutte le transizioni. Durante ogni simulazione si misura lo stato del sistema e, una volta raccolti i dati di ogni iterazione dell'algoritmo si fa una media delle misure. Essa rappresenta la simulazione completa dell'evoluzione del sistema.

L'algoritmo ha complessità pari ad $\mathcal{O}(\boldsymbol{N})$ ad ogni singola iterazione, con $\boldsymbol{N}$ = numeri di transizioni.

Il maggiore svantaggio del metodo Kinetic Monte Carlo è la necessità di conoscere a priori i rate associati ad ogni transazione. Ciò non è possibile per molti sistemi fisici, i quali richiedono un rate che cambia in modo dinamico in base all'evoluzione del sistema.

2.2.2 Algoritmo di Gillespie

L'algoritmo di Gillespie [6] è un'estensione dell'algoritmo Kinetic Monte Carlo, usato in modo pervasivo nella simulazione di sistemi chimici e biologici. Esso, a differenza dei metodi analitici basati su equazioni differenziali usati nella chimica classica, riesce a simulare sistemi formati da un numero di molecole molto basso (anche nell'ordine della decina). Questo è particolarmente importante per la simulazione di reazioni intracellulari, solitamente formate da un numero esiguo di molecole chimiche.

L'algoritmo si basa sull'ipotesi di *sistema ben mescolato* (*well-stirred system*), in cui le molecole sono equamente distribuite sul volume $\boldsymbol{\Omega}$ del contenitore. Per questo tipo di sistemi non è necessario specificare posizione e velocità di ogni molecola per ogni istante di tempo t, ma per caratterizzarlo è sufficiente il vettore $\boldsymbol{X}(t) = (X_1(t), \ldots, X_N(t))$, dove $X_i(t)$ rappresenta il numero di molecole S_i contenute in $\boldsymbol{\Omega}$ al tempo t.
Le reazioni tra molecole possono essere soltanto di due tipi:

- **Reazioni uni-molecolari**, dove una singola molecola cambia forma, ad esempio decomponendosi in altre molecole
- **Reazioni bi-molecolari**, dove sono coinvolte due molecole sia di tipo diverso che uguali.

Tutti gli altri tipi di reazioni (reversibili, trimolecolari) sono costruiti come una serie di reazioni mono o bi-molecolari.

Per un sistema *ben mescolato* una reazione R_j è caratterizzata da due quantità: il vettore di cambio stato $\mathbf{v}_j = (v_{1j}, \ldots, v_{Nj})$ dove v_{ij} è il cambiamento nel numero di molecole S_i causato dalla reazione R_j; la reazione R_j, quindi,

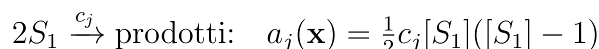
induce il cambiamento di stato $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}_j$. La seconda quantità è definita come *funzione di propensità* (*propensity function*) per la reazione R_j. Essa è definita come:

$$a_j(\mathbf{x})dt \equiv \text{la probabilità che, dato } \mathbf{X}(t) = \mathbf{x} \text{ la reazione } R_j \text{ si verificherà in } \mathbf{\Omega} \text{ nell'intervallo di tempo } [t, t + dt)$$

Per trovare la funzione di propensità per una condizione R_j è necessario conoscerne il rate statico c_j, definito in modo che $c_j dt$ rappresenti la probabilità che una combinazione casuale dei reagenti per la reazione R_j reagiscano tra loro nell'intervallo temporale $(t, t + dt)$. Conoscendo il rate statico per tutte le reazioni può essere calcolata la funzione di propensità nel seguente modo:

$$S_1 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = c_j[S_1]$$

$$S_1 + S_2 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = c_j[S_1][S_2]$$



$$2S_1 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = \frac{1}{2}c_j[S_1]([S_1] - 1)$$

rispettivamente per reazioni a un solo reagente, a due reagenti formati da molecole diverse e con due reagenti dello stesso tipo. $[S_i]$ indica la concentrazione della molecola S_i in $\mathbf{\Omega}$.

Stocastic simulation algorithm

Un sistema modellato nel modo visto fin ora può essere simulato attraverso la *chemical master equation* (CME), che evolve il sistema in modo completamente deterministico. Tuttavia risolverlo attraverso la CME richiede di risolvere un sistema di equazioni differenziali, il che potrebbe non essere immediato.

Un altro approccio al problema è lo *Stochastic simulation algorithm* (SSA), il quale simula in modo stocastico una possibile traiettoria nell'evoluzione del sistema. Realizzando lo SSA diverse volte e facendo una media delle varie simulazioni si ottengono risultati simili a quelli ottenuti risolvendo numericamente la CME.

Per costruire una traiettoria simulata di $\mathbf{X}(t)$ è necessaria una funzione di probabilità definita nel seguente modo:

$$p(\tau, j|\mathbf{x}, t)dt \equiv \text{la probilità dato } \mathbf{X}(t) = \mathbf{x} \text{ che la prossima reazione avvenga nell'intervallo di tempo } [t + \tau, t + \tau + d\tau) \text{ e che sia la reazione } R_j. \tag{2.1}$$

Formalmente questa funzione è l'unione di due funzioni di densità di probabilità per le variabili aleatorie τ e j, rispettivamente tempo e indice della prossima reazione da eseguire.

Si introduce una seconda funzione di probabilità $P_0(\tau|\mathbf{x}, t)$ definita come la probabilità che, con il sistema nello stato $\mathbf{X}(t) = \mathbf{x}$, nessuna reazione di alcun tipo avvenga nell'intervallo di tempo $[t, t + \tau)$. Per la definizione di propensità e e per la teoria della probabilità la funzione P_0 deve soddisfare:

$$P_0(\tau + d\tau|\mathbf{x}, t) = P_0(\tau|\mathbf{x}, t) \times \left[1 - \sum_{j'=1}^{M}(a_{j'}(\mathbf{x})dt)\right] \tag{2.2}$$

dove la prima parte indica la probabilità che nessuna reazione avvenga in $[t, t+\tau)$ e la seconda parte che nessuna reazione avvenga in $[t+\tau, t+\tau+d\tau)$. con $d\tau$ piccolo a sufficienza per contenere una ed una sola reazione. Definendo $a_0(\mathbf{x}) = \sum_{j'=1}^{M}(a_{j'}(\mathbf{x})dt)$ e passando al limite $d\tau \to 0$ si ottiene l'equazione

$$\frac{dP_0(\tau|\mathbf{x}, t)}{d\tau} = -a_0(\mathbf{x})P_0(\tau|\mathbf{x}, t) \tag{2.3}$$

che, risolta per le condizioni iniziali $P_0(0|\mathbf{x}, t) = 1$, è uguale a

$$P_0(\tau|\mathbf{x}, t) = \mathrm{e}^{-a_0(\mathbf{x})\tau} \tag{2.4}$$

La probabilità descritta in (2.1) può essere scritta come

$$p(\tau, j|\mathbf{x}, t)dt = P_0(\tau|\mathbf{x}, t) \times (a_j(\mathbf{x})d\tau) \tag{2.5}$$

poiché la prima parte dell'equazione non è altro che la probabilità che nessuna reazione avvenga in $[t, t + \tau)$, mentre la seconda parte rappresenta la probabilità che la reazione R_j avvenga nell'intervallo $[t + \tau, t + \tau + d\tau)$ Sostituendo $P_0(\tau|\mathbf{x}, t)$ con il risultato trovato in (2.4) si ottiene

$$p(\tau, j|\mathbf{x}, t) = a_j(\mathbf{x})\,\mathrm{e}^{-a_0(\mathbf{x})\tau} \tag{2.6}$$

Che riscritta in un altro modo diventa

$$p(\tau, j|\mathbf{x}, t) = a_0(\mathbf{x})\,\mathrm{e}^{-a_0(\mathbf{x})\tau} \times \frac{a_j(\mathbf{x})}{a_0(\mathbf{x})} \tag{2.7}$$

Si può notare come τ sia una variabile casuale di distribuzione esponenziale, e con media e varianza pari a $1/a_0(\mathbf{x})$, mentre j è una variabile casuale indipendente con probabilità pari a $a_j(\mathbf{x})/a_0(\mathbf{x})$.

Per generare le due variabili in questione esistono molti metodi, tra cui, quello usato nell'algoritmo di Gillespie, è l'estrazione di due numeri casuali r_1 e r_2 nell'intervallo [0, 1], che consentono di generare τ e j nel seguente modo:

$$\tau = \frac{1}{a_0(\mathbf{x})} \ln\left(\frac{1}{r_1}\right) \tag{1.8a}$$

$$j = \text{ il più piccolo intero che soddisfa } \sum_{j'=1}^{j} a_{j'}(\mathbf{x}) > r_2 a_0(\mathbf{x}) \tag{1.8b}$$

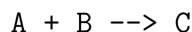
Usando queste formule è possibile costruire un'implementazione dello SSA nel seguente modo:

1. Si inizializza il tempo $t = t_0$ e lo stato iniziale $\mathbf{x_0} = \mathbf{x}$
2. Con il sistema nello stato $\mathbf{x}$ e al tempo t si valutano tutte le funzioni di propensità $a_j(\mathbf{x})$ e la loro somma $a_0(\mathbf{x})$.
3. Vengono generati i valori di τ e j usando le equazioni (1.8a) e (1.8b).
4. Viene eseguita la reazione scelta, viene aggiornato il tempo come $t \leftarrow t+\tau$ e lo stato come $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}_j$
5. Si compie un'osservazione del sistema. Se la simulazione è finita STOP, altrimenti si ritorna allo stato 2.

Implementando più volte l'algoritmo sopra riportato e facendo una media di tutte le misurazioni si ottiene una traiettoria del sistema molto simile a quella che si troverebbe risolvendo il problema in modo analitico.

2.2.3 Ottimizzazioni

L'algoritmo di Gillespie consente di simulare un sistema, anche molto esteso, in modo accurato usando un limitato numero di risorse computazionali. Tuttavia il suo principale difetto consiste nella limitata velocità di simulazione, dovuta al calcolo delle *propensity function* per tutte le reazioni, ad ogni passo dell'algoritmo. Questo può portare ad un degrado generale delle prestazioni del metodo, si preda come esempio il seguente caso:
Durante una simulazione al tempo t viene eseguita una reazione del tipo



```
A + B --> C
```

dove una molecola `A` e una molecola `B` si combinano per formare una molecola `C`. In questo caso soltanto la concentrazione di `A`, `B` e `C` viene cambiata. Si supponga di avere un'altra reazione del tipo

L'algoritmo ricalcolerebbe la propensità anche per quest'ultima reazione, anche se essa non può essere cambiata, poiché la concentrazione dei reagenti è rimasta la stessa. Si immagini un sistema con milioni di reazioni come può essere una cellula. Ad ogni step dell'algoritmo viene sprecato sia tempo che risorse computazionali per ricalcolare dati che non sono cambiati.

Un modo per risolvere questo problema è quello di mantenere in memoria le dipendenze tra reazioni, ovvero, per quali ha senso ricalcolare la propensità dopo l'esecuzione della reazione j. Ciò viene implementato attraverso un grafo $G = (V, E)$, detto grafo delle dipendenze, che ha come vertici le reazioni. Un arco $(i, j) \in E$ se e solo se l'esecuzione della reazione R_i influenza il calcolo della propensità per la reazione R_j. Ovviamente ogni vertice avrà un arco che punta a se stesso. Dopo aver eseguito la reazione R_i, quindi, si dovrà calcolare la *propensity function* solo per quelle reazioni che sono direttamente collegate al vertice R_i. Un modello del grafo delle dipendenze è visibile in figura Figura 2.6.

Un secondo problema dell'algoritmo di Gillespie è la generazione di numeri casuali. La loro estrazione, infatti, non è un'operazione elementare. Come dimostrato in [7] è, anzi, l'operazione più costosa per un simulatore biochimico. L'idea è quindi quella di diminuire le estrazioni di numeri casuali, per poter velocizzare la simulazione. Un primo approccio è quello di generate un *tempo putativo* (*putative time*) per ogni reazione, in base alla sua propensità. Il tempo putativo rappresenta l'istante temporale in cui la reazione sarà eseguita, ed è compreso nell'intervallo $(t, +\infty)$. Avendo questo parametro non è più necessario scegliere in modo casuale la prossima reazione da eseguire: basterà, infatti, ordinarle secondo il loro tempo putativo e selezionare come prossima quella con il *putative time* minore. In [7] per la memorizzazione delle reazioni si usa una struttura dati detta *indexed priority queue*, che altro non è che un *heap binario*. L'estrazione della prossima reazione da eseguire avviene quindi in $\mathcal{O}(1)$, e l'ordinamento della struttura ha complessità $\mathcal{O}(\log n)$. Si è dimostrato che usando una struttura dati simile l'aumento di prestazioni è ragguardevole, rispetto alla scelta in modo casuale della prossima reazione da eseguire.

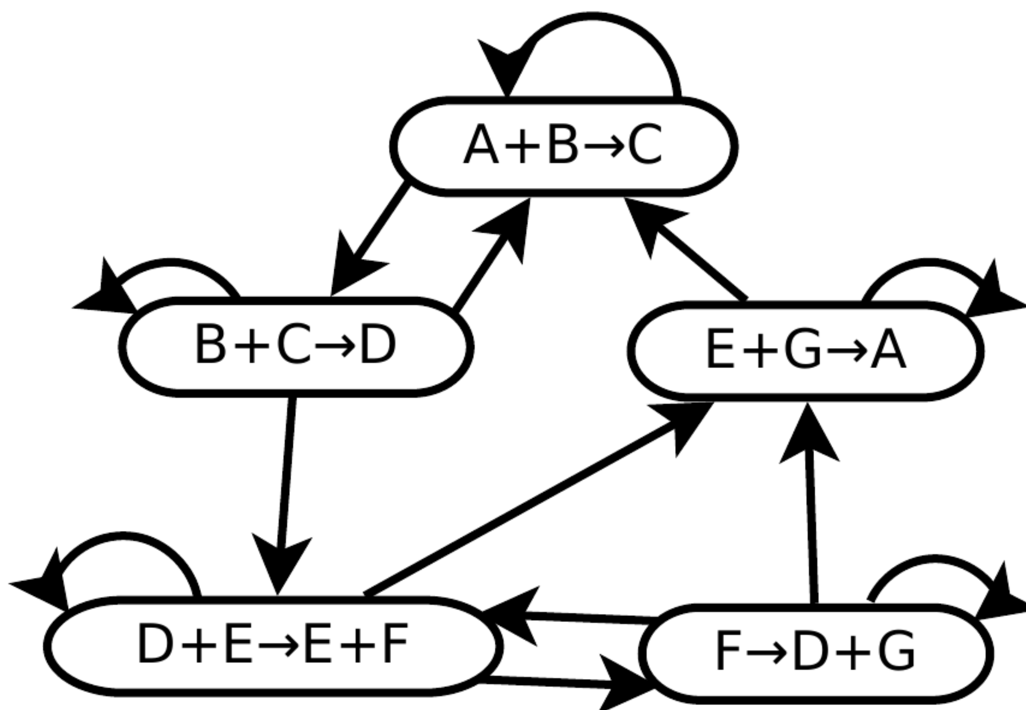


Figura 2.6: Esempio di grafo delle dipendenze per una simulazione. Una reazione R_j ne può influenzare la reazione R_k solo se i reagenti di R_k compaiono in R_j. Notare come ogni reazione influenzi esplicitamente se stessa.

Usando il tempo putativo è possibile ridurre ulteriormente la generazione di numeri casuali, ricalcolando il *putative time* per ogni reazione dipendente da quella appena eseguita usando parametri noti. Il metodo è descritto di seguito:

- Nel caso la reazione R_j di cui si vuole ricalcolare il tempo putativo non è l'ultima reazione eseguita, ma una sua dipendenza:
  - Sia T il tempo corrente della simulazione, τ_c il nuovo tempo putativo, a_c la nuova propensità, τ_p il precedente tempo putativo e a_p la precedente propensità.
  - Posso calcolare $\tau_c = \frac{a_c(\tau_p - T)}{a_p}$.
  - Ciò è possibile solo se $\forall\, \tau_p, T : \tau_p > T$.

Poiché la maggior parte delle reazioni avrà $\tau_p > T$ l'ultima condizione è quasi sempre verificata. Ciò permette di risparmiare pesantemente nella generazione di numeri casuali, velocizzando di conseguenza il motore di simulazione.

Un altro punto critico dello *stochastic simulation algorithm* è il calcolo dell'indice della prossima reazione da simulare. Nella versione originale dell'algoritmo esso viene calcolato sommando progressivamente le propensità delle

reazioni, fino a superare il valore $r_2\, a_0(\mathbf{x})$, con r_2 numero casuale $\in [0, 1]$ e $a_0(\mathbf{x})$ somma di tutte le *propensity function*. Ciò ha un costo computazionale pari a $\mathcal{O}(N)$ con N numero totale di reazioni. Una prima idea per abbassare questo costo è quella di sommare le reazioni da quelle con propensità maggiore a quelle con propensità minore [8]. In questo modo la somma progressiva cresce in modo rapido, superando quindi più velocemente l'indice della reazione obiettivo.

Un approccio migliore al problema è quello proposto da Slepoy *et al.* [9]. Esso si basa sull'idea di scegliere le reazioni usando non uno, ma due numeri casuali. Viene estratto un primo numero $r_1 \in [0, N]$ che rappresenta l'indice della reazione selezionata. Viene estratto un secondo numero random $r_2 \in [0, a_0(\mathbf{x})]$ che rappresenta un valore di propensità casuale. Se la reazione selezionata tramite r_1 ha $a_j(\mathbf{x}) > r_2$ allora scelgo R_j, altrimenti reitero il procedimento. In questo modo la selezione di una reazione è indipendente da N e, se le le reazioni hanno propensità simili, l'algoritmo sarà reiterato un numero molto basso di volte.

Un ulteriore miglioramento è dato dal dividere in gruppi le reazioni in base alla loro propensità. In un gruppo, quindi saranno presenti solo reazioni con un valore della *propensity function* simile. Si procede quindi scegliendo casualmente un gruppo G dall'insieme dei gruppi. Sia $a_p(\mathbf{x})$ la propensità massima di una reazione all'interno di G. Si esegue, quindi, il procedimento descritto in precedenza, usando però $a_p(\mathbf{x})$ come limite superiore al valore di r_2. Avendo tutte le reazioni in un gruppo una propensità simile è molto probabile trovare una reazione che soddisfa le condizioni in un numero molto breve di interazioni.

Si è dimostrato che scegliendo i gruppi nel modo seguente:

- Sia $a_{min}(\mathbf{x})$ la propensità minima.
- Il primo gruppo contiene tutte le reazioni con propensità da 0 a $a_{min}(\mathbf{x})$
- Il secondo gruppo contiene le reazioni con propensità da $a_{min}(\mathbf{x})$ a $2 \cdot a_{min}(\mathbf{x})$
- Il terzo gruppo contiene le reazioni con propensità da $2 \cdot a_{min}(\mathbf{x})$ a $4 \cdot a_{min}(\mathbf{x})$
- In modo analogo si formano tutti gli altri gruppi.

trovare una reazione valida all'interno di un gruppo richiede mediamente meno di due iterazioni dell'algoritmo. Inoltre, scegliendo i gruppi in questo modo, il loro numero è indipendente dal numero totale di reazioni e, anzi, rimane

invariato anche a seguito dell'inserimento di reazioni durante l'esecuzione della simulazione.

Seguendo questo algoritmo si ottiene un costo costante ($\mathcal{O}(1)$) per la scelta della prossima reazione da eseguire, invece del costo di $\mathcal{O}(N)$ dato dallo SSA proposto da Gillespie. Questo miglioramento, unito a quelli trattati precedentemente in questa sezione, consente la simulazione di vaste reti di nodi in modo efficiente e veloce.

Tutti questi algoritmi sono implementati nel motore di simulazione di Alchemist, insieme ad altri aspetti che verranno trattati nella prossima sezione.

2.2.4 Implementazione

Alchemist utilizza tutti i concetti finora espressi per la simulazione di un sistema fisico, ampliandoli e utilizzando una loro generalizzazione. Come detto il simulatore si basa su concetti basati sulla simulazione chimica, come quelli appena esposti, ma anche su concetti propri della *simulazione ad agenti* (*agent based simulation*, ABS). Quest'ultimo tipo di simulatori consente di simulare un'ampio spettro di fenomeni naturali, proprio per la sua natura distribuita. L'idea alla base di Alchemist è, quindi, quella di usare un simulatore basato sul kinetic Monte Carlo, e generalizzarlo per poter simulare un insieme molto vasto di sistemi.

Una prima estensione è quella di rendere il sistema multi-compartimento. Le molecole, quindi, non sono più contenute in un solo contenitore, ma ognuna di esse è posizionata in un nodo, eventualmente comunicante con altri nodi. Per ottenere questo obiettivo è necessario estendere sia il concetto di reazione che quello di grafo delle dipendenze. Una reazione ha un nuovo parametro, chiamato *contesto*, che può assumere tre valori: *local, neighborhood* e *global*. Una reazione ha due contesti: il contesto di input, ovvero quale parte dell'ambiente è letta dalla reazione, e il contesto di output, ovvero quale parte dell'ambiente è modificata dalla reazione. Un contesto *local* significa che solamente il nodo in cui è presente la reazione subirà modifiche. Un contesto *neigborhood* significa che uno o più nodi vicini subiranno modifiche (ad esempio il passaggio di una molecola tra due nodi). Il contesto *global* sta a significare che la reazione influenzerà anche nodi non vicini. Ovviamente più i contesti delle reazioni sono piccoli maggiore sarà la velocità della simulazione, poiché l'esecuzione di una reazione produrrà un ricalcolo della propensità per meno reazioni.

Siano R_1 e R_2 due reazioni, ed R_1 appena eseguita. Si dovrà ricalcolare la propensità di R_2 (ovvero esiste un arco da R_1 ad R_2 nel grafo delle dipendenze) se, oltre ad agire su molecole comuni si verifica almeno una delle seguenti condizioni:

- R_1 ed R_2 appartengono allo stesso compartimento.

- Il contesto di output di R_1 è *global*.
- Il contesto di input di R_2 è *global*.
- Il contesto di output di R_1 è *neighborhood* e R_2 è in un nodo vicino.
- Il contesto di input di R_2 è *neighborhood* e R_1 è in un nodo vicino.
- Il contesto di output di R_1 e di input di R_2 sono entrambi *neighborhood* e c'è almeno un nodo che è sia vicino al nodo di R_1 che a quello di R_2 (i due nodi hanno un vicino in comune).

Applicando queste regola alla costruzione del grafo delle dipendenze si ottiene un *grafo delle dipendenze spaziale*, che consente di simulare ambienti a più compartimenti, come sono gli ambienti multi-cellulari.

Alchemist è un software *open source* e quindi il codice è pubblicamente visibile e modificabile. I sorgenti possono essere consultati al seguente link: `https://github.com/AlchemistSimulator/Alchemist`. Esso si presenta come un progetto formato da un insieme di sotto-progetti, ognuno con un compito specifico. All'interno del progetto `alchemist-engine` si trovano le implementazioni dei concetti visti fino ad ora. La classe `Engine` è il cuore del motore di simulazione. Attraverso il metodo `doStep` viene estratta la prossima reazione da eseguire dalla *index priority queue*, che tiene le reazioni ordinate per tempo putativo. Se non ci sono reazioni da eseguire la simulazione è conclusa. In caso contrario viene eseguita la reazione estratta, e vengono aggiornate le sue dipendenze. Esse sono gestite tramite la classe `DependencyHandlerImpl` che, per ogni reazione R_j memorizza le reazioni influenzate da R_j e quelle che influenzano R_j, secondo il contesto. Il grafo delle dipendenze è implementato nella classe `MapBasedDependencyGraph`, che altro non è che una mappa tra ogni reazione ed il corrispettivo gestore delle dipendenze.

I nodi non sono però fissi in un unico punto per tutta la durata della simulazione, ma possono muoversi, rompendo e formando sempre nuove relazioni di vicinato. La gestione di questo aspetto, già compresa nell'implementazione presentata, permette di simulare ambienti anche molto dinamici, rendendo Alchemist un simulatore ad ampissimo spettro, in grado cioè di simulare i più disparati sistemi fisici.

2.3 Tool di sviluppo

Alchemist è un grande progetto software, e per la sua gestione è necessario ricorrere a tecniche avanzate ed allo stato dell'arte. In particolare, per il suo sviluppo, si è fatto ricorso a un *Distributed Version Control System* (DVCS). In

particolare, Alchemist utilizza git[1] come DVCS, in virtù delle sua prestazioni e della sua leggerezza. Esso consente a più sviluppatori di lavorare al progetto, in totale autonomia. Ogni sviluppatore possiede una copia locale del progetto, su cui vengono apposte delle modifiche. Attraverso l'operazione di *commit* viene aggiornato lo stato del repository locale, salvando i cambiamenti rispetto alla versione precedente. Quando due sviluppatori decidono di unire le loro modifiche eseguono un *merge* tra le due copie. Esso unisce i cambiamenti fatti in modo automatico. Alchemist utilizza un'estensione di git, detta *git-flow*[2]. Essa divide i progetti in più rami, ognuno con un compito specifico. Il ramo principale, detto *master*, contiene la versione stabile del progetto. Ogni volta che è necessaria una modifica al progetto si crea un nuovo ramo, partendo da quello master, e si compiono lì le modifiche. Una volta ultimate, il ramo figlio si riunisce a quello master. In questo modo non si sporca il ramo principale con modifiche continue, che potrebbero anche non andare a buon fine.

Alchemist è ospitato su *github*[3], un sito web che ospita progetti mantenuti con git. Esso funge da repository online per il progetto, con cui tutti gli sviluppatori possono sincronizzarsi.

2.3.1 Continuous integration

La continuous integration (integrazione continua), è una pratica che si applica allo sviluppo software, in particolare di sistemi complessi. Essa si basa sul presupposto che lo sviluppo indipendente di grandi porzioni di software porta a numerosi problemi, il più comune dei quali è il cosiddetto *integration hell*. In questo scenario molti sviluppatori lavorano su parti diverse della stessa applicazione, in modo intensivo e in periodi di tempo lunghi. Venuto il momento di unire i lavori, essi sono così diversi e discordanti che la loro unione impiega più tempo di quello che è stato necessario a sviluppare le modifiche. In scenari negativi si potrebbe dover arrivare a ritornare allo stato di partenza.

La continuous integration consiste nell'integrare presto e spesso il lavoro fatto, eseguendo test e automatizzando i processi di *build*.

Alchemist fa uso intensivo di strumenti di continuous integration, come *gradle* e *drone.io*

[1] `https://git-scm.com`
[2] `http://danielkummer.github.io/git-flow-cheatsheet`
[3] `https://github.com`

Gradle

Uno dei principali problemi di grandi sistemi software è il cosiddetto *dependency hell*[4]. Esso si presenta quando si fa affidamento su una o più librerie esterne che, a loro volta, dipendono da altre librerie. Aggiornare una di esse potrebbe voler dire aggiornare tutte le sue dipendenze, in una catena quasi senza fine che è praticamente ingestibile se il numero di dipendenze supera la decina.

Gradle è un sistema di automazione dello sviluppo e di compilazione automatica del codice, che consente, tra le altre cose, di ovviare a questo problema. Esso, in automatico durante la compilazione, provvede a scaricare le ultime versioni delle librerie necessarie (la scelta può essere configurabile, si può, ad esempio, forzare gradle a scaricare una data versione di una libreria) e di tutte le loro dipendenze.

Durante la compilazione, gradle può eseguite anche i test automatizzati, verificando la correttezza del software ad ogni modifica. In particolare ciò è utile per constatare il corretto funzionamento di vecchie parti del sistema a fronte di modifiche.

Gradle lavora attraverso un insieme di *task*, ognuno con un compito specifico. Ogni task può dipendere da altri task: in questo caso gradle utilizza un *directed acyclic graph* (DAG) per mappare le dipendenze.

Drone.io

Un sistema come Gradle permette di evitare molti dei problemi tipici di una cattiva gestione del software. Tuttavia esistono alcune situazioni in cui eseguire una build sempre sulla stessa macchina può portare a dei problemi, tra cui:

- La cache della macchina su cui si lavora contiene un pacchetto che in realtà non esiste più.
- Alcuni file sono localizzati tramite percorsi assoluti.
- Una risorsa è presente soltanto nella macchina locale e non nel repository online.

In tutti questi casi il progetto sembra correttamente funzionare nella macchina su cui lo si sviluppa, salvo poi non funzionare correttamente sui pc di altre persone. Per questo motivo sarebbe buona norma fare la build del progetto su una macchina pulita.

[4] `https://en.wikipedia.org/wiki/Dependency_hell`

Drone.io è un servizio web che ha il suddetto scopo. Esso è collegato con github, ed ogni volta che rileva cambiamenti nel repository fa partire una build completa del progetto (con anche l'esecuzione dei test) su una macchina virtuale totalmente pulita. Ciò è estremamente vantaggioso per due motivi: la build su macchina pulita consente di individuare problemi altrimenti non facilmente scovabili, come quelli esposti sopra. Inoltre, per progetti molto grandi, come Alchemist, la build completa potrebbe impiegare parecchi minuti per essere completata. Eseguendola online ed in modo automatico non si perde tempo, potendo concentrarsi esclusivamente sullo sviluppo.

Capitolo 3

Analisi dei requisiti

3.1 Requisiti cellulari

In questa sezione verranno trattati ed analizzati i requisiti comuni a tutte le cellule simulate, siano esse all'interno di un gruppo o isolate.

Ogni cellula dovrà essere identificabile, attraverso un valore univoco, all'interno dell'ambiente di simulazione. Essa conterrà al suo interno un insieme di reazioni ed un insieme di molecole chimiche, ognuna con il proprio valore di concentrazione. Una molecola è identificata da un nome. Esso non si limita ad identificare una molecola all'interno di una cellula, ma, se due specie chimiche residenti in due cellule distinte, hanno lo stesso nome esse sono considerate molecole del medesimo tipo. Questo concetto è identico a quello usato in chimica, dove, ad esempio, la sigla CO_2 identifica il biossido di carbonio, in qualunque ambiente esso si trovi. A differenza delle scienze chimiche, in Biochemistry, il nome delle molecole potrà essere scelto a discrezione dell'utente, usando anche sigle *ad hoc*.

3.1.1 Reazioni biochimiche

Le reazioni biochimiche sono caratterizzate da un insieme di condizioni e uno di azioni. Una condizione rappresenta un'affermazione, che deve essere vera affinché la reazione possa essere eseguita. Solo quando tutte le condizioni sono verificate nello stesso momento, la reazione potrà essere schedulata dal sistema, e quindi eseguita. L'insieme delle condizioni può anche essere vuoto. Ciò sta a significare che la reazione è sempre valida, e, quindi, verrà eseguita di continuo. Una condizione può essere un qualsiasi requisito sullo stato del sistema, ad esempio:

- Concentrazione di una molecola nella cellula maggiore o minore di un certo valore.

- Numero di cellule vicine uguale, maggiore o minore ad un dato numero.
- Presenza, o assenza, di un particolare tipo di giunzione cellulare.
- Concentrazione di una molecola, in una cellula vicina, maggiore o minore ad un certo valore.
- Presenza o assenza, nell'ambiente, di una data biomolecola.

Un'azione, al contrario, è una funzione che attua un cambiamento nello stato della simulazione. Una reazione può presentare anche diverse azioni, che saranno eseguite nello stesso istante temporale. L'insieme delle azioni potrebbe anche essere vuoto. Ciò non ha scopi pratici, ma è stato incluso tra i requisiti per mantenere il concetto di reazione il più generico possibile.

Oltre alle condizioni ed alle azioni, una reazione, dovrà essere caratterizzata da un numero $r \in \mathbb{R}_{\geq 0}$ detto *rate* della reazione. Esso rappresenta il concetto di velocità di reazione. Più il rate è alto, più la reazione avverrà velocemente e in un tempo vicino al tempo corrente della simulazione. Questo numero rimane fisso, e per questo è detto *rate statico*. Esso caratterizza la reazione allo stesso modo di condizioni e azioni, e dovrà quindi essere fornito dall'utente.

Le condizioni, inoltre, dovranno fornire un numero reale, maggiore di zero, chiamato *propensità della condizione*. Esso rappresenta l'influenza della condizione nel calcolo della velocità della reazione. Esso è un concetto mutuato dalla chimica, in cui, il calcolo del *reaction rate*, avviene prendendo in considerazione la concentrazione dei reagenti [10].

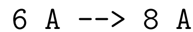
Tutte le reazioni che rispondo ai requisiti qui presentati sono considerate reazioni valide. Ciò può portare ad avere reazioni che, non avendo condizioni, compiono azioni sulla simulazione. Ad esempio si potrebbero creare dal nulla molecole chimiche, e nello stesso modo distruggerle, andando a violare la *legge di Lavoisier*. Ciò deve essere permesso e considerato valido, per permettere la simulazione della reazione del sistema a comportamenti anomali.

3.1.2 Equilibrio delle reazioni chimiche

In accordo con la legge di conservazione della massa, i reagenti di una reazione chimica si combinano tra loro per formare i prodotti, o si scompongono in molecole più semplici, mantenendo inalterata la loro massa. In modo più semplicistico si può dire che i reagenti vengono distrutti e al loro posto sono creati i prodotti della reazione. Ciò è il comportamento che deve avere l'incarnazione Biochemistry con le reazioni chimiche. Tutte le molecole presenti nelle condizioni devono venire distrutte, e al loro posto create le molecole presenti nelle azioni. Questo è il comportamento standard di tutte le reazioni.

Se una molecola chimica, presente nelle condizioni, dovrà essere mantenuta nel sistema, essa dovrà venir specificata anche tra le azioni della stessa reazione. Questo comportamento consente di modellare in modo corretto tutte le reazioni chimiche.

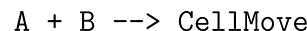
Un caso particolare è dato dalle reazioni che hanno le stesse molecole, sia come reagenti, che come prodotti. Si supponga ad esempio che la reazione:



```
6 A --> 8 A
```

sia eseguita all'interno di una cellula con una concentrazione di `A` pari a 10 molecole. Essa rimuove dalla cellula 6 molecole `A` e ne aggiunge 8. La concentrazione, alla conclusione della reazione dovrà essere $10 - 6 + 8 = 12$ molecole.

Un altro caso particolare è dato dalle reazioni che presentano molecole tra le condizioni, ma non tra le azioni. Un esempio di tale reazione è esposto di seguito:



```
A + B --> CellMove
```

dove, se le molecole `A` e `B` sono presenti, viene attuato il movimento della cellula. In questo caso, le due molecole presenti come reagenti, vengono eliminate. Dopo la reazione, quindi, la loro concentrazione sarà ridotta di una unità, all'interno della cellula.

3.2 Requisiti multi-cellulari

In questa sezione vengono discussi i requisiti necessari alla corretta simulazione di un ambiente multi-cellulare.

3.2.1 Vicinato

Il primo concetto, e forse il più importante, è proprio quello di vicinanza tra due cellule. Esse si dicono vicine se la loro distanza è tale da poterle considerare a contatto. Le cellule dovranno avere, quindi, una posizione, data come coordinate cartesiane, che rappresenta il punto dell'ambiente in cui sono posizionate. Prendendo due cellule, e conoscendone la posizione, è possibile calcolarne la distanza. Se essa è minore a una data distanza minima, passata come input alla simulazione, le cellule sono considerate adiacenti. È necessario, quindi, che l'ambiente in cui sono immerse le cellule abbia un'estensione spaziale, bi o tridimensionale.

Un altro concetto importante, che deriva da quello di distanza, è quello di vicinato (o *neighborhood*). Esso rappresenta l'insieme delle cellule che hanno

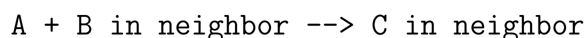
una distanza minore della distanza minima, rispetto ad una cellula in esame. Il calcolo del vicinato è quindi eseguito a partire da una cellula, detta *centro*, e restituisce tutte le cellule adiacenti a quella centrale. Solo quelle appartenenti allo stesso vicinato possono essere coinvolte in reazioni dirette tra cellule, come lo scambio di molecole o la creazione di giunzioni.

3.2.2 Reazioni che coinvolgono il vicinato

Le reazioni possibili in Biochemistry non si limitano a quelle intracellulari. Devono essere possibili, infatti, anche reazioni tra cellule adiacenti, come lo scambio diretto di molecole, o il compimento di azioni in base a condizioni sul numero o la vicinanza di cellule adiacenti.

Le reazioni con il vicinato si presentano del tutto simili, nei requisiti, a quelle meramente intracellulari. Nel caso ci siano più condizioni, però, esse devono essere soddisfatte tutte dalla stessa cellula. Se ha una reazione richiede che una molecola del tipo `A` ed una del tipo `B` siano presenti in un vicino, ed esse sono presenti in due cellule adiacenti distinte, essa è considerata non valida.

Stesso discorso può essere fatto per le azioni che riguardano cellule adiacenti, dividendo però in due casi questa situazione. Nel caso siano presenti condizioni sui vicini, come detto, devono essere tutte verificate nella stessa cellula, che prende il nome di *bersaglio*. Se, oltre alle condizioni, ci sono anche azioni che modificano lo stato di cellule adiacenti, la cellula su cui saranno eseguite quelle azioni sarà il bersaglio. Ad esempio si consideri la seguente reazione:



```
A + B in neighbor --> C in neighbor
```

dove, se le molecole `A` e `B` sono presenti in una cellula adiacente, crea una molecola `C` in una cellula vicina. In questo caso la cellula dove sarà posizionata la molecola `C` sarà la stessa, se esistente, che contiene una molecola `A` e una `B`. Si ricorda, inoltre, che quanto detto per le reazioni intracellulari vale anche per le reazioni con il vicinato. I due reagenti saranno quindi eliminati, e al loro posto sarà creata una molecola `C`. Nel caso più di un vicino soddisfi tutte le condizioni, la scelta della cellula bersaglio non avverrà in modo casuale tra tutte quelle valide, ma tramite una funzione discussa in fase di design.

Un caso particolare di quanto fin'ora detto è dato dalle reazioni che non hanno condizioni sul vicinato, ma hanno azioni su di esso. In questo caso non è possibile scegliere una cellula bersaglio a partire dalle condizioni. Essa dovrà quindi essere scelta in modo casuale, tra tutte quelle adiacenti alla cellula su cui è eseguita la reazione.

3.2.3 Reazioni che coinvolgono l'ambiente

Come visto nel primo capitolo, le cellule comunicano tra loro attraverso il rilascio di molecole nell'ambiente, che saranno poi rilevate dalle cellule vicine, o anche molto lontane nel caso di segnalazione a distanza. Dovrà essere possibile, quindi, la simulazione di reazioni che rilasciano, o ricevono, molecole dall'ambiente. Un problema di questo requisito è dato dal fatto che Alchemist non supporta la presenza di molecole libere nell'ambiente. Inoltre le specie chimiche rilasciate dalle cellule dovrebbero avere un'estensione spaziale, dato che devono, al pari delle cellule stesse, potersi muoversi liberamente nell'ambiente. Per questi motivi, il concetto di reazione con l'ambiente extra-cellulare, non è stato trattato in questo progetto, ma sarà oggetto di lavori ed estensioni future.

3.3 Requisiti sulle giunzioni

Una importante parte di questo progetto è stata focalizzata sulla costruzione delle giunzioni tra cellule. In questa sezione vengono presentati ed analizzati i requisiti che esse devono soddisfare all'interno dell'incarnazione Biochemistry.

3.3.1 Creazione di giunzioni

Una giunzione è "un'unione" tra due cellule, che risultano quindi legate. Per creare una giunzione sono necessarie tre condizioni:

- Le due cellule devono essere vicine.
- Una o più biomolecole devono essere presenti nella cellula corrente.
- Una o più biomolecole devono essere presenti nella cellula vicina.

Se tutte queste condizioni sono valide la giunzione è correttamente creata. Le molecole usate per la sua creazione non possono essere usate per altre reazioni. Esse rimangono all'interno delle cellule, ma fanno parte della giunzione. Le condizioni fin'ora esposte sono necessarie, ma potrebbero non essere sufficienti. Certi tipi di giunzioni, infatti, potrebbero richiede la presenza di altre condizioni, come la presenza nell'ambiente di una data molecola.

La creazione di una giunzione deve poter essere rilevata dalle cellule, in modo che esse possano scatenare altre reazioni intracellulari, note come catene. Il numero di giunzioni che una cellula può instaurare non deve essere limitato, così come il numero di vicini distinti con cui instaurare giunzioni. Due cellule possono creare tra loro un numero molto alto di giunzioni, anche dello stesso tipo.

3.3.2 Distruzione di giunzioni

Le giunzioni non sono eterne, al contrario, esse vengono continuamente create e distrutte dalle cellule. Dovrà quindi essere possibile distruggere una giunzione. Quando un evento del genere accade, le molecole che formavano la giunzione devono ritornare nelle cellule di origine. Se, ad esempio, tra due cellule si è formata una giunzione tramite le molecole: `A` presente nella prima cellula e `B` nella seconda, al momento della distruzione della giunzione, `A` dovrà ritornare a far parte della prima cellula, mentre `B` della seconda.

3.3.3 Comunicazione attraverso giunzioni

Tramite le giunzioni deve essere possibile anche la comunicazione diretta tra cellule (si pensi ad esempio alle *gap junctions*). Dovrà quindi essere possibile scrivere reazioni che, verificata la presenza di un particolare tipo di giunzione, scambino molecole tra le due cellule, come se tra loro fosse presente un passaggio. Questo tipo di comunicazione è simile a quello visto per le reazioni che coinvolgono il vicinato, ma ha una condizione in più: la presenza della giunzione in esame. In questo caso il nodo bersaglio sarà quello dall'altro capo della giunzione. Il caso in cui siano presenti più giunzioni dello stesso tipo sarà analizzato in fase di design.

Non tutte le giunzioni portano ad una comunicazione diretta tra le cellule. Sarà quindi l'utente che, in base al tipo di giunzione, specificherà l'eventuale comunicazione.

Lo schema di tutti i requisiti cellulari espressi in questo capitolo è visibile in Figura 3.1.

3.4 Interfacciamento con il sistema

L'interfacciamento con il sistema dovrà essere tale da permettere agli utenti di inserire facilmente tutti i dati necessari al corretto set-up di una simulazione. Dovrà anche fornire meccanismi con cui i dati generati saranno reperibili in modo semplice e intuitivo.

Alchemist, per la scrittura delle simulazioni, utilizza YAML (*YAML Ain't a Markup Language*), un linguaggio di markup semplice e leggibile. Per molti aspetti, come il posizionamento delle cellule, questo approccio è ottimale. In un modo simile al linguaggio naturale, si possono compiere azioni anche complesse e iterate su più oggetti. Le reazioni biochimiche, però, sono specifiche per l'incarnazione Biochemistry, e anche la loro grammatica dovrà essere *ad hoc.*

Dovrà essere possibile visualizzare, in modo grafico, l'insieme di cellule che si stanno simulando, con effetti grafici che evidenziano la concentrazione delle

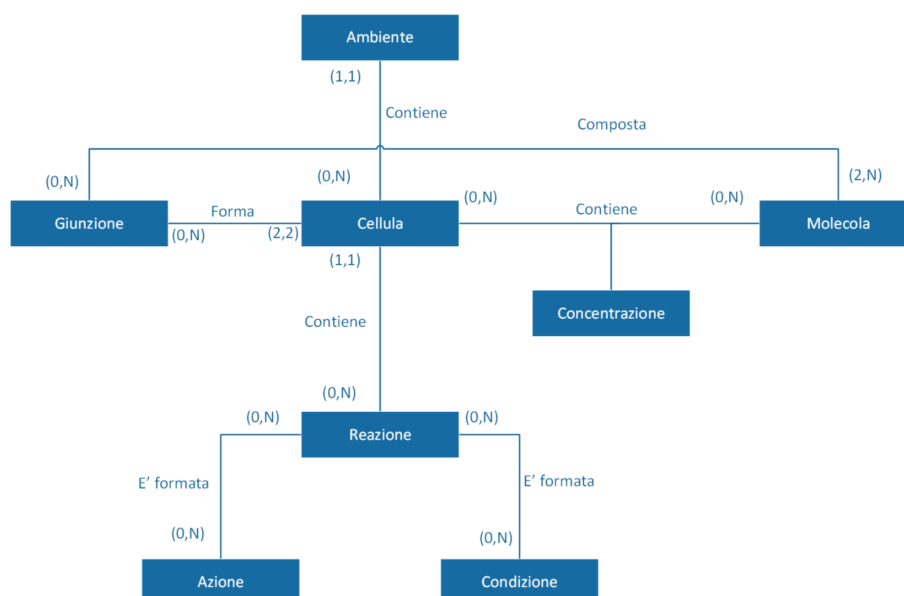


Figura 3.1: Schema dei componenti del dominio applicativo, illustrati nei requisiti. L'ambiente contiene un numero indefinito di cellule, che a loro volta possono contenere molecole e reazioni. All'interno di una cellula, ad ogni molecola è associata una concentrazione. Le reazioni sono formate da azioni e condizioni. Il loro numero può anche essere pari a zero. Una cellula può formare giunzioni con altre cellule, a patto che esse facciano parte dello stesso vicinato. Una giunzione è formata da almeno due molecole, una per cellula.

varie biomolecole. Dovrà essere anche possibile monitorare, in tempo reale, la concentrazione delle molecole chimiche all'interno delle cellule. Allo stesso modo dovranno essere mostrate le giunzioni.

La velocità della simulazione potrà essere modulata, in modo che sia possibile rallentarla fino a poche reazioni eseguite al secondo. In questo modo sarà possibile visualizzare in modo chiaro i cambiamenti che esse producono nell'ambiente.

3.4.1 Specifica delle reazioni chimiche

Le reazioni chimiche dovranno essere specificate in un modo simile a quanto usato nella chimica e nella biologia. Le condizioni dovranno essere espresse nella parte sinistra della reazione, mentre le azioni in quella destra. Ogni condizione ed ogni azione dovrà avere un *contesto*, che indica la sua esecuzione nella cellula corrente, in una vicina o nell'ambiente. Le reazioni dovranno presentare una grammatica coincisa, in grado cioè di rappresentare reazioni a molte condizioni e molte azioni in poche righe di testo.

Le molecole chimiche potranno avere qualsiasi nome, a patto che inizi con un carattere alfabetico, e che contenga al suo interno soltanto caratteri alfanumerici ed underscore. La concentrazione delle molecole chimiche dovrà essere un numero reale positivo.

Dovrà, inoltre, essere possibile specificare condizioni e azioni *custom*, ovvero già implementate in apposite classi Java, all'interno dell'incarnazione.

L'ordine delle condizioni e delle azioni non dovrà essere importante, anche cambiandolo la reazione dovrà produrre gli stessi risultati. Se dovessero essere presenti errori sintattici nella specifica di una reazione, il simulatore dovrà mostrare un messaggio di errore, ed interrompere il caricamento del programma.

Reazioni con giunzioni

La specifica delle reazioni che coinvolgono giunzioni varia in base alla posizione della giunzione nella reazione. Se si tratta di una reazione di creazione la giunzione dovrà essere presente solo nella parte destra (tra le azioni). Tra le condizioni devono necessariamente esserci almeno una condizione su una molecola nella cellula, e una su una molecola in una cellula vicina. Se queste due condizioni non dovessero essere presenti, la reazione sarebbe mal scritta, e dovrà produrre un messaggio di errore.

La distruzione delle giunzioni dovrà avvenire in modo implicito. Se una giunzione è presente nella parte sinistra della reazione, ma non in quella destra,

essa, al pari delle altre molecole, viene eliminata. Notare come ciò comporti l'immissione nelle due cellule delle biomolecole che formavano la giunzione.

Se una giunzione è presente sia a destra che a sinistra essa non verrà eliminata. È il caso delle reazioni di comunicazione tra giunzioni, in cui essa rimane anche dopo l'esecuzione. Se a destra, nella reazione, ci sono giunzioni che non esistono tra le condizioni allora la reazione è malformata, e dovrà essere notificato all'utente, tramite un messaggio di errore.

3.4.2 Risultati delle simulazioni

Le simulazioni dovranno produrre risultati, sotto forma di dati numerici, rappresentanti la concentrazione di una o più molecole contenute all'interno di una o più cellule. Questi dati dovranno essere campionati in tempi periodici prestabiliti. Dovrà essere possibile, inoltre, aggregare i dati delle simulazioni, come ad esempio computare la media delle concentrazioni di una molecola nella cellule, o la sua varianza. Sarà, inoltre possibile, durante la simulazione stessa, visualizzare parametri come il numero di iterazioni fatte, ed il tempo totale simulato.

Capitolo 4

Design

In questo capitolo verrà trattato il progetto dell'incarnazione Biochemistry. Verranno quindi discusse le scelte progettuali e di design, che hanno portato alla sua implementazione.

4.1 Mappa dei requisiti su Alchemist

Alchemist, come trattato nel secondo capitolo, usa diverse astrazioni per rappresentare i concetti propri di ogni sistema. Esse sono estremamente generiche, in modo da consentire la modellazione di sistemi molto diversi tra loro.

Una parte importante della progettazione dell'incarnazione è il giusto *mapping* tra concetti prettamente biochimici, in astrazioni usate dal simulatore Alchemist. Tra i requisiti di Biochemistry, c'è la modellazione dei seguenti concetti:

- Cellula.
- Molecola chimica.
- Reazione, composta da reagenti (condizioni) e prodotti (azioni).
- Giunzioni tra cellule adiacenti.

Per la sua ispirazione biochimica, la mappatura di questi concetti sul simulatore è risultata semplice e lineare. Non tutte le astrazioni, però, sono nativamente presenti e supportate da Alchemist. In questi casi si è dovuta operare una precisa scelta di design, in modo da poter modellare questi concetti, senza sconvolgere l'architettura di base del sistema.

Figura 4.1: L'interfaccia Incarnation. Gli argomenti dei metodi non sono riportati per semplicità.

La classe principe da implementare è l'estensione dell'interfaccia *Incarnation* (vedi Figura 4.1). Essa è usata da Alchemist, durante la fase di caricamento della simulazione, per creare tutte le altre astrazioni. Ogni incarnazione, infatti, usa una diversa implementazione delle interfacce presenti nel modello del simulatore. La prima scelta progettuale da compiere è scegliere il tipo della concentrazione. Anche se il nome potrebbe far pensare che essa sia sempre un numero, in realtà esistono casi in cui la concentrazione delle molecole deve rappresentare una diversa astrazione. Nel caso di Biochemistry la concentrazione delle molecole nei nodi dovrà essere un numero reale, e quindi modellato come un `Double`.

Si è costruita un'estensione dell'interfaccia *Incarnation*, chiamata `BiochemistryIncarnation`, la quale ha lo scopo di produrre, attraverso i metodi ereditati dall'interfaccia, tutti concetti modellati di seguito.

4.1.1 Molecole

Alchemist gestisce nativamente il concetto di molecola, attraverso l'interfaccia *Molecule* (vedi Figura 4.2). Essa è totalmente generica, e rappresenta una qualsiasi entità sia contenuta all'interno di un nodo. Infatti presenta solo due metodi: `getId`, che restituisce l'identificativo della molecola, e `dependsOn`, che calcola la dipendenza della molecola da quella data come parametro.

Figura 4.2

In Biochemistry, una molecola è definita soltanto dal nome. Tra le classi già implementate dal simulatore, c'è anche la modellazione di base di una molecola, tramite classe astratta. Essa estende l'interfaccia *Molecule*, aggiungendo la gestione del nome, come unico campo di rilievo. Si è quindi estesa la classe astratta `SimpleMolecule`, tramite la classe `Biomolecule`, senza particolari step di progettazione.

4.1.2 Nodi

Le cellule sono forse l'entità più complessa da mappare su un simulatore. Esse sono molto difficili da implementare, sia per la loro complessità strutturale, sia per le migliaia di organelli e molecole contenute al loro interno.

<<Interface>>
Node
addReaction(Reaction)
cloneNode()
contains(Molecule)
getChemicalSpecies()
getConcentration(Molecule)
getContents()
getId()
getReactions()
hashCode()
removeConcentration(Molecule)
removeReaction(Reaction)
setConcentration(Molecule, T)

Figura 4.3: L'interfaccia Node.

Anche se, come si è detto nel primo capitolo, le cellule presentano una struttura interna molto diversificata, si è deciso di considerarle come corpi cavi, al cui interno sono contenute molecole chimiche, con una certa concentrazione. Esse sono considerate in sospensione, come se fossero disperse in un gas, con concentrazione costante in tutta la cellula. Seguendo questo approccio è possibile mappare direttamente le cellule su nodi Alchemist. Un nodo è modellato attraverso l'interfaccia *Node* (visibile in Figura 4.3). Essa presenta, tra gli altri, i metodi per ricevere la concentrazione di una data molecola (anche in questo caso la concentrazione è considerata generica), sapere se una molecola è contenuta nella cellula e settarne la concentrazione. Anche le reazioni vengono gestite, con metodi che consentono il loro inserimento ed eliminazione dalla cellula. Come si potrà notare, non ci sono metodi per la gestione delle giunzioni. Esse, infatti, non sono un concetto modellato nativamente dal simulatore Alchemist, e la loro implementazione merita una sezione a parte (vedi Sezione 4.1.4).

Tuttavia, la progettazione delle cellule, deve prevedere la gestione delle giunzioni. Si è deciso, quindi, di modellare un'interfaccia, denominata *ICellNode*, che estende *Node*. Essa dovrà fornire un contratto, che, tutte le cellule devono soddisfare. Le funzionalità progettate sono:

- Restituzione di tutte le giunzioni presenti in un nodo, ognuna con l'indicazione della cellula vicina con cui è connessa

- Aggiunta di una giunzione alla cellula, dato in ingresso il tipo di giunzione e la cellula adiacente.

- Rimozione di una giunzione, con la reimmissione delle molecole formanti la giunzione stessa, nella cellula in esame.

- Un metodo che, data in ingresso una giunzione, restituisca se essa è presente nella cellula.

- Restituzione del numero totale di giunzioni presenti nella cellula in esame.

- Dato in ingresso un tipo di giunzione, restituire l'insieme dei nodi legati a quella cellula con la giunzione data. Se la cellula non presenta al suo interno il tipo di giunzione dato come input, dovrà restituire un insieme vuoto.

- Restituzione di tutte le cellule, collegate a quella in esame, tramite qualunque tipo di giunzione.

Questi requisiti sono molto simili a quelli che un nodo deve soddisfare rispetto alle molecole.

Una cellula, in Biochemistry, è stata modellata attraverso la classe `CellNode`. Essa deve implementare tutti i metodi sia dell'interfaccia *Node*, sia di *ICellNode*. In Alchemist è presente una classe astratta chiamata `GenericNode`, che gestisce in modo semplice reazioni e molecole. La classe `CellNode`, quindi, estende il nodo generico, aggiungendovi l'implementazione e la gestione delle giunzioni cellulari. Uno schema delle classi UML dove è visibile questa modellazione è visibile in Figura 4.6.

4.1.3 Reazioni

Le reazioni sono uno dei concetti di maggiore complessità all'interno del simulatore Alchemist. Come detto, esse sono formate da condizioni ed azioni. In Alchemist esse sono modellate attraverso l'interfaccia *Reaction* (vedi Figura 4.4). Tra i metodi più importanti ci sono il metodo `getInfluencedMolecules`, che restituisce le molecole influenzate, ovvero quelle presenti tra i prodotti della reazione. Esso, insieme al metodo `getInfluencingMolecules`, che, al contrario, ritorna le molecole che influenzano l'esecuzione della reazione: i suoi reagenti, permette di costruire il grafo delle dipendenze, in unione con i contesti di input e output. Un altro metodo particolare è `update`. Esso viene chiamato dalla simulazione ogni qualvolta la reazione deve aggiornare la sua *propensity function*, ovvero ogni volta che essa è eseguita o è appena stata eseguita una delle sue dipendenze. Ciò significa verificare la validità di tutte le condizioni, e calcolare la propensità della reazione in base al suo rate ed alla *propensity function* di ogni condizione. Il metodo `update`, quindi, si dovrà occupare di valutare tutte le condizioni. Se almeno una di esse è falsa la reazione non potrà eseguire. Le condizioni dovranno, inoltre, produrre un numero reale, usato per calcolare la propensità totale della reazione.

<<Interface>>
Reaction
canExecute()
execute()
getInfluencedMolecules()
getInfluencingMolecules()
getInputContext()
getOutputContext()
getRate()
update()

Figura 4.4: L'interfaccia Reaction. Sono riportati solo i metodi più significativi.

Calcolo del rate

Un problema in questo approccio è il calcolo del rate. Sappiamo, dal Capitolo 2, che in Alchemist una reazione è formata da condizioni, azioni, e un

numero reale, detto *rate statico* della reazione. Si ricorda che, secondo l'algoritmo di Gillespie, la *propensity function* di una reazione deve essere calcolata nel seguente modo:

$$S_1 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = c_j[S_1]$$

$$S_1 + S_2 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = c_j[S_1][S_2]$$

$$2S_1 \xrightarrow{c_j} \text{prodotti:} \quad a_j(\mathbf{x}) = \tfrac{1}{2}c_j[S_1]([S_1]-1)$$

dove c_j rappresenta il rate statico e $[S_i]$ la concentrazione della biomolecola S_i. Queste equazioni rappresentano esattamente il coefficiente binomiale tra, il numero di molecole richieste dalla condizione e la concentrazione di esse nella cellula. Infatti si ha che:

$$\binom{n}{1} = n \qquad \text{e} \qquad \binom{n}{2} = \frac{1}{2} \cdot n \cdot (n-1)$$

che sono esattamente le formule usate teoricamente dall'algoritmo di Gillespie. Ogni condizione dovrà avere un metodo che restituisce un intero pari al coefficiente binomiale tra concentrazione della molecola nella cellula su numero di molecole richieste. Notare come $\binom{0}{n} = 0$, per cui, se la molecola in esame non è presente nella cellula, la propensità sarà zero, e la reazione non eseguita, come voluto.

In accordo con quanto definito da Gillespie, questi numeri reali dati dalle condizioni dovranno essere moltiplicati tra loro (nel caso ci sia più di una condizione) e per il rate statico della reazione. Facendo ciò si ottiene una buona stima della velocità totale della reazione. Questo perché il rate è tanto maggiore quanto sono maggiori le concentrazioni delle molecole. E più molecole sono presenti, maggiore sarà la probabilità che due di esse si incontrino, producendo la reazione.

Nel caso ci siano condizioni che non coinvolgono molecole, come, ad esempio: numero di cellule vicine maggiore di 3, la propensità della condizione dovrà essere valutata caso per caso.

Condizioni

Le condizioni devono quindi avere, principalmente, due metodi. Il primo, denominato `isValid`, quando chiamato controllerà la validità della condizione, e restituirà un valore booleano atto ad indicarla. Il secondo, denominato `getPropensityConditioning`, restituirà un numero reale, calcolato nel modo visto sopra. Ciò è implementato nell'interfaccia *Condition*, la quale è estesa da una folta gerarchia di classi, ognuna rappresentante una particolare condizione.

Le condizioni implementate sono:

`BiomolPresentInCell` condizione sulla presenza di una biomolecola in una cellula, ad una data concentrazione.

`BiomolPresentInNeighbor` condizione sulla presenza di una biomolecola in una cellula adiacente a quella in cui è presente questa condizione. Anche in questo caso è dato in ingresso il valore minimo di concentrazione.

`JunctionPresentInCell` verifica la presenza di una giunzione, data in input, nella cellula.

`JunctionPresentInNeighbor` verifica la presenza della giunzione data in input all'interno di una cellula adiacente. Notare che la giunzione deve collegare la cellula in cui è contenuta questa condizione.

`NeighborhoodPresent` verifica la presenza di almeno un vicino della cellula corrente.

Azioni

Le azioni, modellate attraverso l'interfaccia *Action*, posseggono un metodo chiamato `execute`, che, come suggerisce il nome, ha lo scopo di eseguire l'azione specifica. Come per le giunzioni una grande varietà di azioni diverse compone l'albero di ereditarietà, che ha come radice l'interfaccia *Action*.

Le azioni implementate sono:

`ChangeBiomolConcentrationInCell` data in ingresso una molecola, e un delta di concentrazione, cambia la concentrazione della molecola nella cellula corrente. Essa potrebbe salire o scendere in base al delta. Se si raggiunge lo zero, la biomolecola non deve più essere considerata contenuta dalla cellula.

`ChangeBiomolConcentrationInNeighbor` come sopra, con l'unica differenza che questa azione viene compiuta su una cellula vicina (se esistente).

`AddJunctionInCell` aggiunge una giunzione alla cellula corrente, avente come cellula collegata un nodo passato in ingresso.

`AddJunctionInNeighbor` aggiunge una giunzione in una cellula adiacente (se esistente). La giunzione aggiunta collegherà il nodo vicino scelto, con la cellula che contiene questa azione.

`RemoveJunctionInCell` rimuove la giunzione passata in ingresso dalla cellula, che collega la cellula stessa e un nodo passato come input. Se non esiste una giunzione corrispondente ai parametri non fa nulla.

`RemoveJunctionInNeighbor` come sopra, con la differenza che l'azione è compiuta su una cellula vicina data come parametro. Essa rimuove la giunzione tra quella cellula e quella su cui quest'azione è eseguita.

4.1.4 Giunzioni

Quelli visti fino ad ora sono concetti già presenti in Alchemist, e la loro modellazione non ha richiesto particolari tecniche, se non quella di mappare i requisiti su astrazioni già presenti. Al contrario, le giunzioni, sono componenti non previsti dal simulatore standard, e la loro progettazione ha richiesto uno studio più profondo.

Come visto nel Capitolo 1, le giunzioni possono essere di tre tipi: di ancoraggio, occludenti, o comunicanti. Si è deciso, tuttavia, di modellare le giunzioni come un unico concetto, senza differenza tra i vari tipi. Questo perché saranno presenti reazioni che coinvolgono direttamente le giunzioni, permettendo di modellarne il comportamento a discrezione dell'utente. Sarà quindi l'utilizzatore del software che deciderà quali giunzioni permetteranno una comunicazione, e quali, invece, scateneranno una cascata di reazioni all'interno delle cellule collegate, quando esse sono create.

Per la modellazione delle giunzioni sono stati proposti vari approcci, tra cui:

Creazione di nodi giunzione tra i due nodi cellulari Questo approccio consentiva la creazione di nuovi nodi tra due cellule legate da una giunzione, chiamati *nodi giunzione*. Essi erano diversi dai nodi cellula ordinari, e contenevano al loro interno le molecole appartenenti alle due cellule, in modo da poterle reimmettere nelle stesse una volta spezzata la giunzione, ed eliminato il relativo nodo. Questa idea presenta però diverse problematiche, tra cui la continua creazione e distruzione di nodi, operazione che potrebbe rallentare di molto la simulazione, e l'uso del concetto di nodo in modo errato.

Uso di una *linking rule* per la modellazione della giunzione In questo approccio una giunzione è modellata attraverso una regola di collegamento tra nodi. Due cellule risultano quindi collegate se e solo se presentano una giunzione comune. Questa idea, però, risulta inapplicabile, sia perché sarebbe necessario l'uso di una linking rule per ogni tipo di giunzione, sia perché due cellule possono avere, tra loro, lo stesso tipo di giunzione più volte, e questo non verrebbe rilevato.

Modellazione di una giunzione come una molecola In questo approccio una giunzione è modellata come uno speciale tipo di molecola, contenuta

all'interno dei nodi. Ogni cellula conterrà quindi una lista di tutte le giunzioni formate, con le rispettive cellule collegate. Questa idea è preferibile, ed è quella usata, sia per la maggiore semplicità implementativa, sia per l'utilizzo corretto di tutte le astrazioni di Alchemist.

Si è scelto quindi di modellare le giunzioni come particolari tipi di molecole. I metodi usati dalle cellule per la loro gestione sono già stati discussi nella Sezione 4.1.2, quindi ora si passerà alla progettazione delle giunzioni vere e proprie.

Esse, come le molecole, saranno caratterizzate da un nome, sotto forma di stringa, che identifica il tipo di giunzione. Una giunzione è formata da due istanze della classe progettata, ognuna inserita in una cellula che compone il legame. Una giunzione dovrà, quindi, memorizzare al suo interno le molecole chimiche che la formano: in particolare dovrà memorizzare quali molecole appartengono alla cellula in cui è contenuta l'istanza, e quali appartengono alla cellula collegata. Non è possibile utilizzare una sola istanza di giunzione all'interno di una sola cellula, poiché, in questo caso, solo una delle due cellule che formano la giunzione sarebbe al corrente della sua presenza.

Si è deciso che le giunzioni non possono avere, a differenza delle biomolecole, nomi a totale discrezione dell'utente. Esse dovranno avere un nome del tipo: `A:2B-C`, dove il segno "`-`" divide le molecole delle due cellule, mentre il segno "`:`" divide molecole della stessa cellula. Un nome come quello in esempio sta a significare che la giunzione è formata da una molecola `A` e due `B`, appartenenti alla cellula in cui è inserita, e da una molecola `C` nella cellula vicina. Ovviamente tra le due cellule il nome sarà invertito. Per una migliore comprensione del concetto vedere la Figura 4.5.

le giunzioni sono state modellate attraverso la classe `Junction`, che implementa l'interfaccia *Molecule*. Essa dovrà fornire metodi che restituiscano il nome, le molecole che formano la giunzione nella cellula corrente, e quelle che la formano nella cellula collegata.

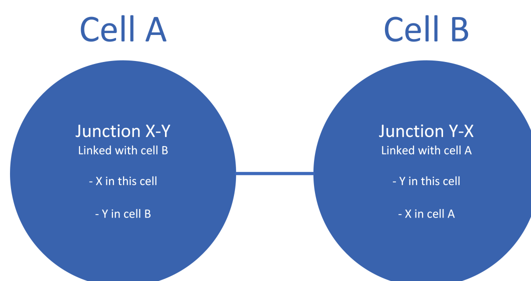


Figura 4.5: Schema di una giunzione tra due cellule. La giunzione è formata da due molecole: la molecola `X`, che apparteneva alla cellula A, e la molecola `Y`, che apparteneva alla cellula B. Ambedue le cellule hanno un'istanza della classe `Junction` al loro interno. L'istanza contenuta in A dovrà avere un nome uguale ad `X-Y`, come molecole contenute nella cellula correte `X`, e come molecole contenute nella cellula collegata `Y`. L'istanza della giunzione in B dovrà essere totalmente speculare. Dovrà quindi chiamarsi `Y-X`, e avere come molecole contenute nella cellula correte `Y`, e contenute in quella collegata `X`. Notare come entrambe le giunzioni abbiano un riferimento alla cellula da loro collegata.

4.2 Reazioni con cellule vicine

Alchemist supporta nativamente le reazioni tra cellule vicine, ma non nel modo corretto per l'incarnazione Biochemistry. Infatti, per come sono strutturate le condizioni, esse non devono essere soddisfatte tutte dallo stesso nodo. Ad esempio, in una reazione del tipo:

```
A + B in neighbor --> whatever
```

le molecole `A` e `B` potrebbero essere contenute in vicini diversi.

4.2.1 Unione delle condizioni

Si rende necessaria, quindi la cosiddetta *unione delle condizioni*, ovvero tutte le condizioni comprendenti cellule vicine devono essere soddisfatte tutte nello stesso nodo.

Per risolvere questo problema è stata modellata una classe astratta, denominata `AbstractNeighborCondition`, che dovrà essere implementata da tutte le condizioni sul vicinato. Essa possiede un metodo astratto, `getValidNeighbors`, il quale, dati in ingresso un insieme di nodi, ritorna il sottoinsieme di essi che

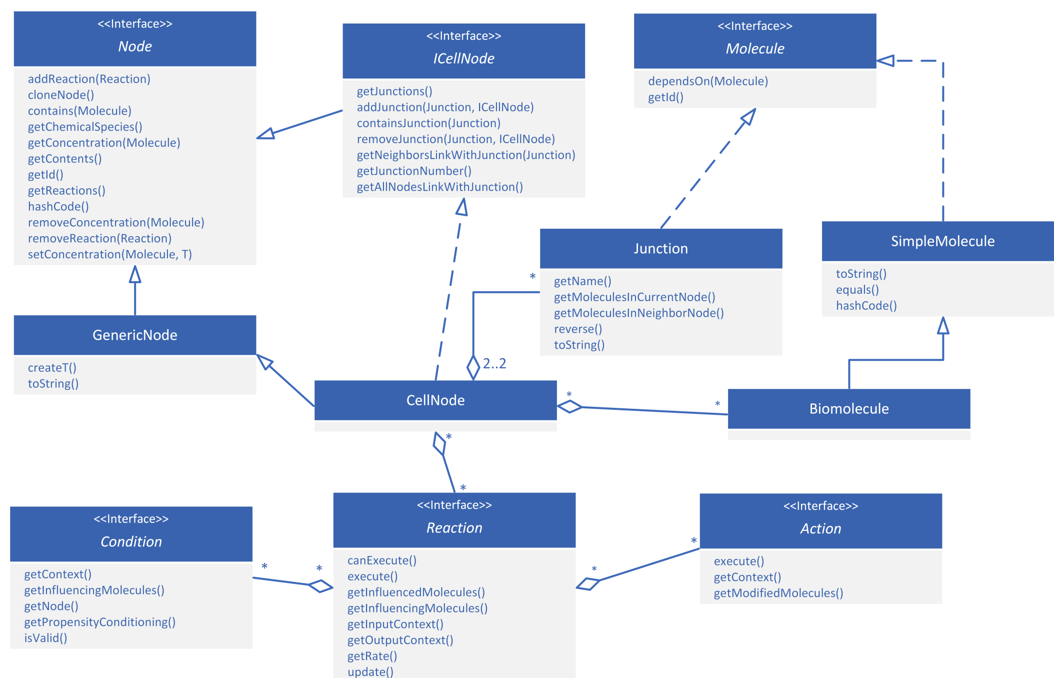


Figura 4.6: Schema UML delle classi del dominio applicativo descritto fino ad ora. Non sono riportati, per semplicità, i metodi ereditati dalle superclassi o dalle interfacce e i campi delle classi. L'interfaccia *Node* descrive il comportamento tipico di una cellula, ad eccezione delle giunzioni. Esse sono modellate attraverso l'interfaccia *ICellNode*. Le cellule vere e proprio sono modellate dalla classe `CellNode`, che, oltre ad ereditare direttamente da *ICellNode*, eredita anche da `GenericNode`, una classe già presente in Alchemist, che attua dei comportamenti di base, per la gestione di un nodo. Una cellula è composta da un insieme di reazioni, a loro volta formate da condizioni ed azioni. Questi concetti sono stati modellati attraverso le interfacce *Reaction*, *Condition* e *Action*. Le molecole sono modellata attraverso l'interfaccia *Molecule*. Anche in questo caso è già presente nel simulatore una classe che ne fornisce una implementazione di base, `GenericMolecule`. La classe `Biomolecule`, usata per modellare molecole biochimiche, estende i concetti della suddetta classe. Le giunzioni sono considerate molecole (poiché risiedono dentro ai nodi), e per questo la classe `Junction` implementa l'interfaccia *Molecule*. Una cellula può avere molte giunzioni, ma un'istanza di giunzione può collegare solo due cellule.

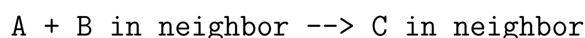
soddisfa la condizione. Se nessun nodo viene passato in ingresso, il controllo viene fatto su tutto il vicinato della cellula. Esso è un'uso particolare del pattern *strategy*, in cui ogni condizione implementa una diversa strategia di filtraggio dei nodi passati in ingresso (vedi Figura 4.7).

La classe `BiochemicalReaction` dovrà, quindi, fare *override* del metodo `updateInternalStatus`, chiamato ogni volta che la reazione viene eseguita, o viene eseguita una delle sue dipendenze. Se sono presenti condizioni sui vicini si dovrà produrre una lista di vicini validi (che soddisfano tutte le condizioni). Si parte quindi con la lista di tutte le cellule adiacenti, e si danno in pasto alla prima condizione sul vicinato. Essa restituirà un sottoinsieme delle cellule vicine che soddisfano quella condizione. Questa lista sarà usata come input per le eventuali altre condizioni sul vicinato. Se la lista, in un qualsiasi momento, si presenta vuota, nessuna cellula soddisfa tutte le condizioni, e perciò la reazione non può essere eseguita. Se, invece, dopo aver computato tutte le condizioni sui vicini, la lista dei nodi validi contiene almeno un nodo, la reazione è valida, ed è possibile eseguirne le azioni.

4.2.2 Selezione del vicino

Trattate le condizioni sul vicinato cellulare, è il momento di trattare le azioni sul vicinato. Esse modificano lo stato di cellule adiacenti a quella in esame, ad esempio cambiando la concentrazione di una molecola, o attuando il movimento del nodo.

In Alchemist, un'azione su un vicino viene eseguita su un nodo adiacente scelto in modo casuale, e ciò non è il comportamento voluto. Supponiamo sia presente la reazione:



```
A + B in neighbor --> C in neighbor
```

dove, se sono presenti le molecole `A` e `B` in un vicino, esse si uniscono a formare la molecola `C`. Non avrebbe senso immettere `C` in un nodo che non contiene `A` e `B`.

Come visto nella precedente sezione, se sono presenti condizioni sul vicinato, la cellula è a conoscenza dell'insieme di nodi che le soddisfano tutte. Soltanto questi nodi potranno subire le azioni. In qualunque modo si scelga uno dei nodi validi, la reazione è formalmente corretta. Non è però giusto scegliere a caso. Ad esempio, prendendo in considerazione la reazione precedente, se due cellule la soddisfano, ma la prima ha una concentrazione di `A` e `B` pari a 10^9 molecole, mentre la seconda ha solo una molecola di ogni specie, l'azione sarà infinitamente più probabile che accada sulla prima cellula. Non è sufficiente, quindi, la sola lista dei nodi validi, ma ad ogni nodo dovrà essere

associato un valore di *propensità*. Come nodo bersaglio per le azioni si dovrà scegliere il nodo a propensità maggiore.

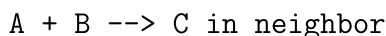
Il calcolo della propensità è del tutto simile a quello visto per le condizioni intracellulari, nella Sezione 4.1.3. Se due nodi presentano la stessa propensità la scelta sarà fatta in modo casuale tra quelle due cellule.

Si è modellata una classe, chiamata `AbstractNeighborAction`, che contiene un metodo astratto `execute(Node targetNode)`, il quale esegue l'azione sul nodo passato come parametro. Se nessun nodo è dato in input l'azione viene eseguita su un vicino a caso (se la cellula possiede un vicinato). Anche in questo caso si è usato il pattern *strategy*, dove ogni implementazione di un'azione attua la strategia definita dalla classe astratta.

In `BiochemicalReaction`, il metodo `execute` non si limiterà ad eseguire tutte le azioni. Esso, infatti, se sono presenti azioni sul vicinato estrarrà, dalla lista dei nodi validi, quello a propensità maggiore, detto *bersaglio* (si ricorda che se la lista è vuota, la reazione non è eseguita). Alle azioni sul vicinato sarà passata in ingresso la cellula bersaglio, modellando in modo accurato anche questo tipo di reazioni.

4.2.3 Casi particolari

Un caso particolare, delle reazioni descritte fin'ora, è dato da quelle che hanno azioni sul vicinato, ma nessuna condizione su di esso. In questo caso la cellula vicina su cui eseguire l'azione deve essere scelta a caso. Questo però può portare a comportamenti indesiderati, come descritto di seguito. Si supponga di avere la reazione:



dove se sono presenti le molecole `A` e `B` nella cellula, esse si combinano a formare una molecola `C`, che viene creata in un vicino. Se la cellula che contiene questa reazione possiede al suo interno i due reagenti, tutte le condizioni sono soddisfatte e la reazione può essere eseguita. Se però la cellula non ha vicini, l'azione non può essere eseguita, ponendo la reazione nel caso di validità delle condizioni, ma impossibilità di eseguire. Per gestire questo caso, si è scelto di aggiungere alla reazione una ulteriore condizione, chiamata `NeighborhoodPresent`, che è vera se e solo se la cellula ha almeno un vicino. Essa viene aggiunta in modo automatico alla reazione, durante la sua creazione, se si rilevano azioni sui vicini, ma non condizioni su di essi. Questa scelta di design permette di sfruttare nel modo migliore le astrazioni di Alchemist, evitando di reinventare la ruota, o attuare scelte in disaccordo con l'architettura generale del sistema.

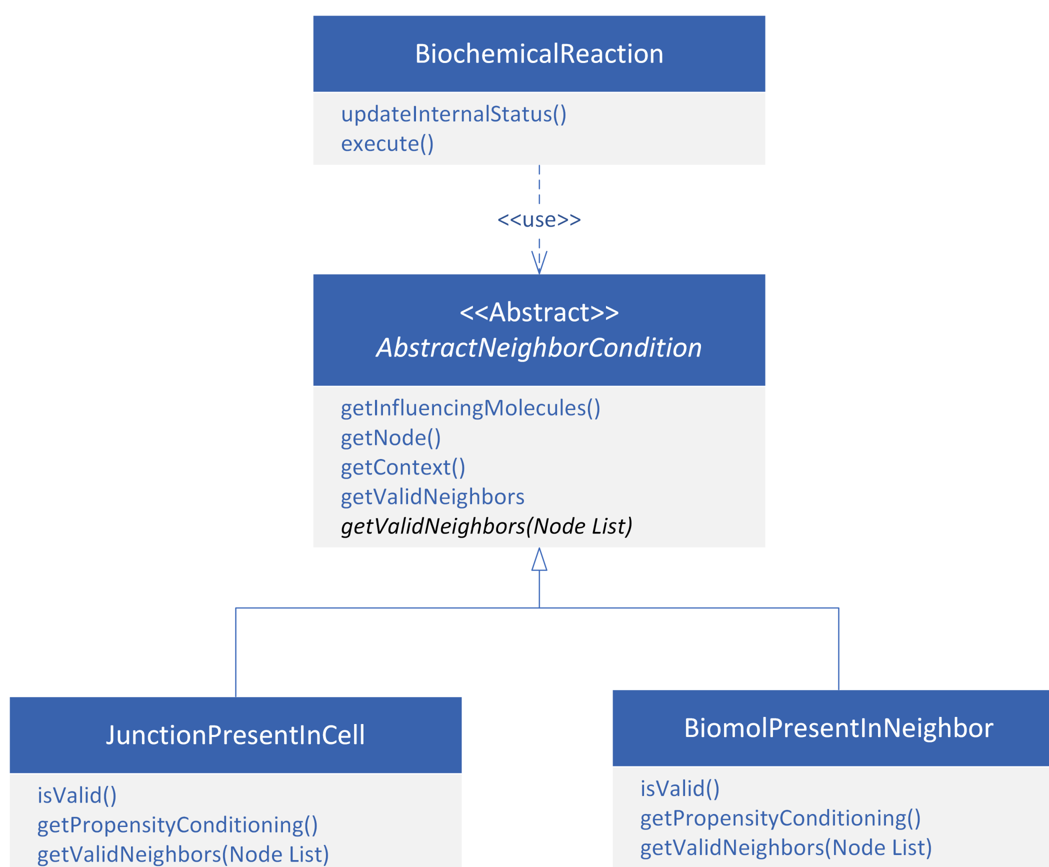


Figura 4.7: Schema UML delle classe che mostra la modellazione dell'unione delle condizioni, nel caso di condizioni sul vicinato. Si è progettata una classe astratta, `AbstractNeighborCondition`, la quale presenta un metodo astratto: `getValidNeighbors(Node List)`. Esso, presa in ingresso una lista di nodi, ritorna un sottoinsieme della stessa, che contiene solo i nodi che soddisfano la condizione. Ogni condizione implementa la propria versione del metodo, che filtrerà i nodi secondo regole diverse.

4.3 Giunzioni cellulari

La modellazione delle giunzioni cellulari è già stata trattata nella Sezione 4.1.4; in questa sezione sono trattati maggiori dettagli progettuali, inerenti alla loro creazione e distruzione.

4.3.1 Creazione

Come detto, una giunzione è uno speciale tipo di molecola, che risiede nelle due cellule collegate (vedi Figura 4.5). Essa, al suo interno, contiene la lista delle molecole che la compongono, sia appartenenti alla cellula in cui è posizionata, sia a quella collegata. In fase di creazione delle giunzione sarà, quindi, necessaria l'immissione di due istanze della classe `Junction`, una per ogni cellula coinvolta. Si ricorda, inoltre, che le due istanze della giunzione, presenti nelle due cellule, devono essere speculari, come mostrato in Figura 4.5.

L'inserimento delle giunzioni nelle due cellule potrebbe essere fatto con una sola azione, la quale inserisce la giunzione nella cellula in cui è schedulata ed in quella collegata. Tuttavia si è deciso di mantenere due azioni distinte, una che posiziona la giunzione nella cellula sulla quale è eseguita la reazione, ed una che posiziona la giunzione, invertita, nella cellula collegata. Questo per non violare il concetto di azione, ovvero di un *singolo* cambiamento fatto all'ambiente di simulazione. L'uso di due azioni è trasparente all'utente. Esso scriverà la reazione semplicemente come:

```
A in cell + B in neighbor --> junction A-B
```

sarà il sistema interno che inserirà le due azioni corrispondenti alla creazione della giunzione.

Le due azioni sono implementate nelle classi `AddJunctionInCell` e `AddJunctionInNeighbor`, ambedue azioni che coinvolgono il vicinato.

4.3.2 Distruzione

La distruzione di una giunzione, oltre a rimuovere la giunzione stessa, deve reimmettere le molecole da cui essa era formata nelle cellule di origine. Come si è visto la giunzione sa quali molecole appartenevo ad una cellula e quale ad un'altra. Non è quindi necessario che sia l'utente a specificare, nelle reazioni di distruzione della giunzione, quali molecole immettere nelle cellule. Si è scelto quindi un approccio implicito, dove una reazione del tipo:

```
junction A-B -->
```

distrugge la giunzione, e reimmette le biomolecole nelle cellule.

Questo comportamento è stato modellato attraverso le sotto-classi di *Action*: `RemoveJunctionInCell` e `RemoveJunctionInNeighbor`. Anche in questo caso si è deciso di dividere la distruzione della giunzione in due azioni distinte.

Sia per la creazione che per la distruzione di giunzioni si è usata l'astrazione di *azione con il vicinato* (vedi Sezione 4.2).Questo perché un'istanza della classe `Junction` rappresenta un *tipo* di giunzione, e non la singola giunzione tra due cellule particolari. Essa, quindi, non contiene dati inerenti al nodo da unire. Sarà compito della reazione, quindi, scegliere tra i nodi vicini validi quello a propensità maggiore, e passarlo, insieme al tipo di giunzione, all'azione corrispondente, tramite in metodo `execute(Node)`, ereditato dalla classe astratta `AbstracNeighborAction`.

Questo è valido sopratutto per la distruzione di giunzioni. In una reazione simile a quella vista sopra, non si ha controllo di quale giunzione viene rimossa. Se la cellula ha, ad esempio, due giunzioni `A-B`, una con la cellula α e una con la cellula β, quale delle due giunzioni rimuovere è scelto a caso.

4.4 Movimento

Il movimento delle cellule è uno degli aspetti più importanti nella modellazione di sistemi multi-cellulari. Quasi tutte le cellule possono muoversi, sia in modo autonomo, come quelle dotate di *flagello*, sia facendosi trasportare, come i globuli rossi. Una sua corretta implementazione è quindi necessaria, al fine di simulare in modo corretto gli ambienti multicellulari.

4.4.1 Vicinato con movimento cellulare

Se le cellule sono libere di muoversi, ad esempio seguendo un *moto browniano*, le reazioni di vicinato potrebbero essere spezzate e ricostruite di continuo. Le cellule adiacenti ad un dato nodo sono quindi variabili nel tempo, e possono cambiare sia a causa del movimento del nodo, sia dei suoi vicini. Nel caso non sia presente movimento, il calcolo dei nodi adiacenti ad una cellula può essere fatto una sola volta, con notevole risparmio di tempo computazionale. Al contrario, se il vicinato è variabile, esso dovrà essere ricalcolato ogni qualvolta è necessario conoscere i vicini del nodo.

4.4.2 Problemi

Un vicinato dinamico porta a diversi problemi alle reazioni tra cellule adiacenti. Per motivi di performance, infatti, l'azione di movimento cellulare ha

contesto *local*, ovvero porta al nuovo calcolo della propensità per le sole reazioni interne al nodo mosso. I vicini della cellula, quindi, non sono notificati del suo movimento, e potrebbero considerarla ancora adiacente, anche se essa ha una distanza maggiore a quella necessaria per la relazione di vicinato. Ciò è dovuto alla modellazione delle le reazioni con cellule vicine (vedi Sezione 4.2). Ogni condizione su una cellula vicina, attraverso il metodo `getValidNeighbors`, restituisce un sottoinsieme della lista di nodi passata in ingresso, formata dai nodi che soddisfano la condizione. Se ci sono più condizioni sui vicini alla prima viene passata la lista di tutte le cellule adiacenti a quella corrente, essa produrrà un insieme di nodi che sono nel vicinato e soddisfano la condizione 1. Passando questa lista alla seconda condizione si otterrà un insieme di cellule che sono vicine e soddisfano le condizioni 1 e 2. Reiterando questo procedimento per tutte le condizioni sul vicinato si ottiene la lista dei vicini validi, da cui si estrarrà il nodo bersaglio, usato dalle eventuali azioni sui vicini. Questo procedimento viene compiuto ogni qualvolta viene chiamato il metodo `updateInternalStatus` della classe `BiochemicalReaction`, ovvero ogni volta che è eseguita la reazione o una delle sue dipendenze. Il movimento di un vicino, però, non è tra le dipendenze, e quindi lo stato interno della reazione non viene aggiornato. Ci si potrebbe trovare, quindi, in situazioni in cui, nella lista dei vicini validi, ci sono nodi che non sono più nel vicinato della cellula. Questa situazione è mostrata in Figura 4.8.

Per risolvere questo problema si sono progettate diverse strategie. La prima, e più ovvia, sarebbe quella di sfruttare il concetto proprio di condizione valida. Una condizione su un vicino è valida se almeno una cellula adiacente la soddisfa. Questo però non risolve il problema (vedi Figura 4.8), poiché molti nodi nel vicinato potrebbero soddisfare la condizione. Se, ad esempio, due cellule soddisfano tutte le condizioni, e quella con propensità maggiore esce dal vicinato, un controllo del genere darebbe comunque esito positivo.

Un'idea migliore, sarebbe quella di porre il contesto del movimento come *neighborhood*, ma ciò rallenterebbe in modo consistente la simulazione.

La soluzione adottata è la seguente: considerando il fatto che per l'algoritmo di Gillespie, adottato da Alchemist, una reazione può avere al più due reagenti, le condizioni sui vicini saranno al massimo due. Le condizioni terranno in memoria la lista dei nodi che le soddisfano. Nel metodo `isValid` controlleranno che tutte le cellule in quella lista siano ancora valide (appartengono al vicinato e soddisfano la condizione). Se anche solo una di esse non è valida la condizione è non soddisfatta. Dato che la maggior parte delle reazioni hanno una sola condizione sui vicini, questo metodo si è rilevato risolutivo, poiché consente di evitare i problemi dati dal vicinato, facendo pochi controlli su di esso.

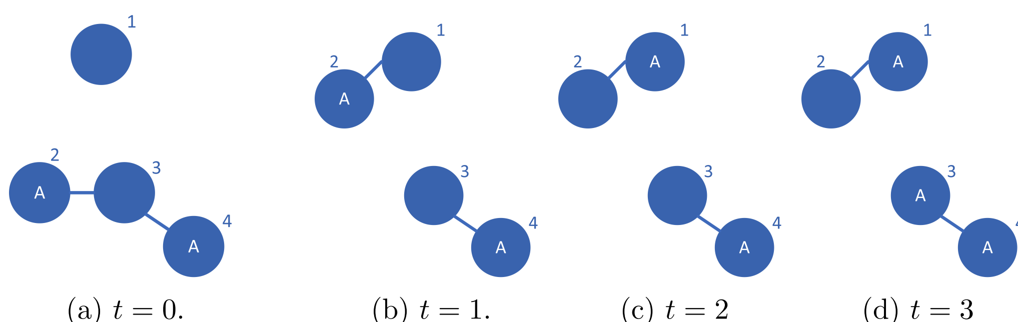
Un altro problema dovuto alle reazioni con vicinato unite al movimen-

to porta alla moltiplicazione delle molecole, in modo incontrollato. Esso si presenta quando, nelle cellule, è presente una reazione del tipo:

```
A in neighbor --> A
```

che sposta una molecola `A` dal vicino alla cellula corrente. Si supponga di avere tre cellule, α, β e γ, con α contenente una molecola `A` e vicina a β, mentre γ isolata. Se prima che β trasferisca la molecola, α si sposta, rompendo il vicinato con β e formandolo con γ, β non viene notificata, e ha ancora la cellula α tra quelle valide. Nel frattempo γ, avendo una cellula con una molecola `A` vicina viene schedulata per eseguire, prima di β. Quando la molecola passa da α a γ, β non viene notificata, poiché è isolata dalle altre due cellule. Trasferisce quindi una molecola da α a se stessa, pur non essendo più adiacenti e la molecola `A` non più presente in α. Questo perché la cellula β non è mai stata notificata dei cambiamenti avvenuti. Si veda la Figura 4.8 per questo comportamento. Ciò comporta una moltiplicazione delle molecole, producendo un comportamento non valido ai fini di una simulazione di sistemi reali.

Il metodo progettato per risolvere il primo problema si è rivelato valido anche in questo caso, risolvendo i problemi emersi in fase di analisi e testing del software.



(a) $t = 0$. (b) $t = 1$. (c) $t = 2$ (d) $t = 3$

Figura 4.8: Le cellule hanno le reazioni di vicinato mostrate in figura. Tutte le cellula presentano al loro interno la reazione: `A in neighbor --> A`, ovvero se esiste almeno un vicino che possiede una molecola `A`, essa viene spostata nella cellula che esegue la reazione. Al tempo $t = 0$ solo la reazione presente nella cellula 3 può essere schedulata, e il suo tempo di esecuzione viene posto a $t = 3$, con bersaglio = cellula 2. Al tempo $t = 1$ la cellula 2 si muove, e rompe il vicinato con la cellula 3. Essa, però, ora è adiacente alla cellula 1, che viene notificata di questo evento. La reazione in 1 può essere schedulata, e il tempo di esecuzione viene posto a $t = 2$. Notare come la cellula 3 sia ignara di tutto ciò, poiché non è più nel vicinato di 2. Al tempo $t = 2$ viene eseguita la reazione in 1, che preleva `A` da 2 e la posiziona su se stessa. La cellula 3 è ancora ignara di tutto ciò. Al tempo $t = 3$ viene eseguita la reazione in 3, poiché essa non è stata mai aggiornata (nessuna dipendenza ha fatto scattare il suo aggiornamento). La cellula 3 crede ancora che 2 sia un nodo valido, e quindi sposta da 2 a se stessa una molecola di `A`. Questa reazione elimina `A` da 2 (che non ha molecole, quindi l'azione non ha effetto), e aggiunge una molecola `A` all'interno del nodo, aumentandone il numero. Si è partiti con due molecole `A` e al tempo $t = 3$ ne sono presenti tre. Notare come anche il controllo che almeno un vicino abbia `A` non modificherebbe la situazione, perché il nodo 4 è ancora adiacente a 3, e possiede una molecola `A`.

4.4.3 Rottura di giunzioni

Il movimento cellulare impatta fortemente sul modello nel caso siano presenti anche le giunzioni. Infatti, tra le condizioni di esistenza di una giunzione, c'è quella per cui le due cellule devono essere adiacenti. Questo concetto è stato modellato con quello di vicinato, e, come si è visto il movimento cellulare può romperlo. Nella fase di progettazione di questo aspetto, si ha un solo requisito: due cellule unite da giunzione devono essere vicine. Si deve adattare, quindi, il movimento a questo requisito.

Si è quindi progettata un'estensione dell'interfaccia *Environment*, la quale è deputata a gestire il movimento. Si è scelto di progettare un ambiente con

margini (*bounds*), poiché le cellule sono di solito contenute in ambienti ristretti. Si è mantenuta comunque la possibilità di avere un ambiente con estensione infinita.

Si sono studiati tre approcci al problema, i quali sono:

- **Le cellule non possono muoversi se presentano giunzioni.** Questo è sicuramente l'approccio di più facile implementazione. Prima del movimento di una cellula si controlla se essa presenta giunzioni. Se il loro numero è maggiore di zero il movimento della cellula non è attuato.
- **Le cellule si muovono, facendo muovere di conseguenza anche tutte quelle collegate con giunzioni.** Questa idea è simile ai sistemi reali, dove le cellule collegate da giunzioni si muovono insieme. Tuttavia non è di facile progettazione. Se una cellula si muove in una direzione d, si dovranno muovere, nella stessa direzione, tutte le cellule ad essa collegate con giunzioni. A loro volta il procedimento dovrà essere reiterato per ogni cellula mossa. Ciò potrebbe comportare il movimento di un numero altissimo di cellule nello stesso momento. Inoltre, le cellule già mosse non dovrebbero essere mosse di nuovo. Servirebbe, quindi, un meccanismo che consenta di sapere, durante l'esecuzione di un movimento, quali nodi sono già stati mossi verso d.
- **Se una cellula si muove, e rompe il vicinato con una cellula con cui ha giunzioni, esse vengono rotte** Modellando il movimento in questo modo, le giunzioni vengono rotte e ricreate in modo molto veloce, simulando ciò che avviene nei sistemi biologici reali. Ogni qualvolta un nodo si muove, si controlla se contiene giunzioni. Se esse sono presenti, e il nodo collegato non è più nel vicinato, la giunzione viene rotta e le molecole che la formavano reimmesse nelle cellule di origine.

Per modellare il movimento, nel caso siano presenti giunzioni cellulari, si è scelto l'ultimo approccio. I nodi sono liberi di muoversi in qualunque direzione, a patto che, se una giunzione si trova nella condizione di collegare due nodi non più vicini, essa viene rotta. Questo comportamento è modellato attraverso la classe `BioRect2DEnvironment`, visibile in Figura 4.9.

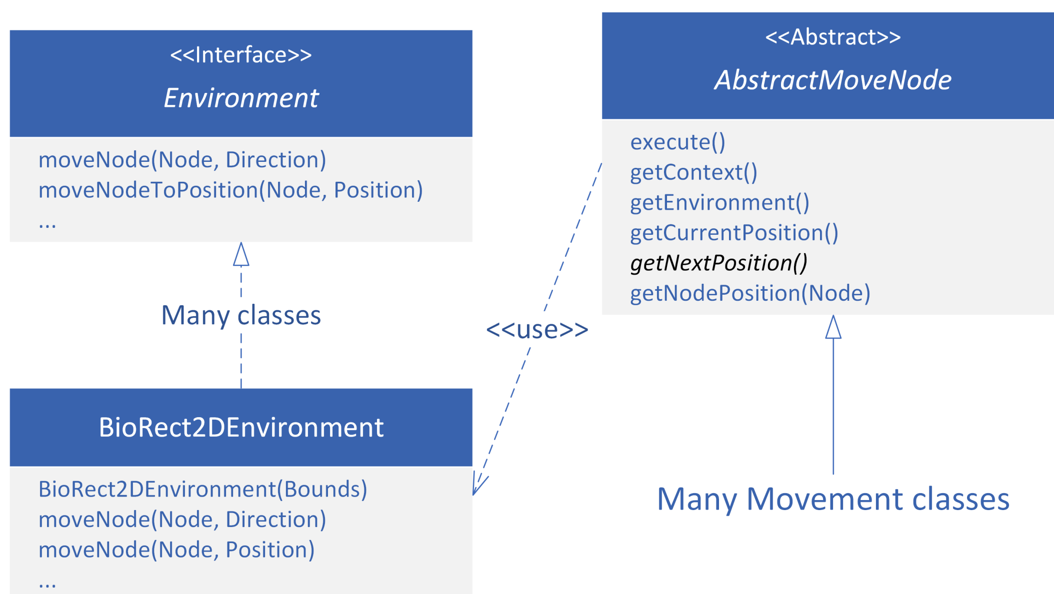


Figura 4.9: Diagramma delle classi UML che mostra la progettazione del movimento cellulare. Si è prodotta un'estensione di *Environment* (le classe intermedie sono state omesse), che modella un ambiente con margini (*bounds*). Essa eredita dall'interfaccia i metodi `moveNode(Node, Direction)`, che muove un nodo verso una direzione, e `moveNode(Node, Position)`, che muove una cellula in una posizione prestabilita. Oltre a controllare che il nodo non oltrepassi i margini dell'ambiente questi due metodi sono usati per gestire la rottura delle giunzioni, se il nodo mosso viene a trovarsi nella situazione di avere giunzioni con cellule non nel vicinato. Tutte le classi che implementano movimento cellulare, in Biochemistry, usano la classe `BioRect2DEnvironment` per attuare il movimento, attraverso il metodo `execute` della classe astratta `AbstractMoveNode`.

4.5 Linguaggio Biochemistry

Come analizzato tra i requisiti, per la scrittura delle simulazioni, Alchemist utilizza YAML, nella sua sintassi standard, tranne che per la scrittura delle reazioni. Una reazione, infatti dovrà avere una sintassi diversa in base al tipo di incarnazione. Si è quindi modellato un DSL (*Domain Specific Language*) specifico per la scrittura di reazioni biochimiche, all'interno dell'incarnazione Biochemistry.

Un DSL è un semplice linguaggio, che si basa su una grammatica. Anche se molto semplice esso deve sottostare ad un insieme di regole per essere considerato valido. Partendo da una stringa, contenente la reazione scritta

dall'utente essa viene passata ad un *Lexer*. Esso si occupa della sua analisi lessicale, ovvero della rilevazione di errori nel lessico di una reazione. Esso, inoltre, spezzetta la stringa in un insieme di *Token*. Ogni token rappresenta un lessema, ovvero un'entità indivisibile alla base del linguaggio, come una parola chiave o una stringa. Dopo questa fase, l'insieme di token viene passato ad un *Parser*, che attua un'analisi sintattica, e produce un albero sintattico (*syntax tree*), della reazione data in ingresso. Durante queste due fasi, sulla base della grammatica, possono essere generati errori di vario tipo, se la reazione data in ingresso non rispetta le regole del linguaggio. L'albero sintattico può essere visitato, partendo dalla radice, in modo da "scannerizzare" la reazione, e crearla all'interno del simulatore. Per visitare il *syntax tree* si è fatto uso del design pattern *visitor* (vedi Figura 4.10), che permette la visita di una struttura dati, incapsulata al di fuori di essa. Ogni nodo dell'albero è visitato, fino ad arrivare alle foglie, generando così una reazione biochimica completa.

Per la costruzione della reazione si è fatto uso del design pattern *builder* (vedi Figura 4.10), che consente di creare la reazione step-by-step. Ciò è utile, poiché la creazione di reazioni chimiche non è un compito immediato, e spezzettarlo in più fasi semplifica di molto il lavoro, e la pulizia del codice.

La classe `BiochemicalReactionBuilder` si occupa della creazione delle reazioni chimiche, date in ingresso dall'utente. Essa ha al suo interno una classe privata statica. Essa implementa il visitor per l'albero sintattico, generato dalla classe builder. Per la gestione degli eventuali errori nelle stringhe delle reazioni si è implementato il pattern *observer*. Esso, modellato attraverso la classe `BiochemistryParseErrorListener`, viene "attaccato" al parser delle reazioni, e rileva eventuali errori, notificandoli all'utente con un messaggio.

I dettagli implementativi di queste classi sono lasciati al capitolo 5.

4.5.1 Costrutti fondamentali

Definito come costruire il DSL, il prossimo passo da fare è quello di progettare la sua grammatica. Una reazione, come più volte detto, è formata da condizioni ed azioni. Esse, in accordo con quanto è d'uso nella chimica, sono state divise da una freccia. si è scelto il simbolo `-->` per la loro separazione: a sinistra le condizioni, a destra le azioni.

La trattazione, si concentra ora sulle reazioni chimiche, ovvero quelle che coinvolgono solo molecole. Una volta definite le regole per questo tipo di reazioni, si estenderanno anche a condizioni ed azioni *custom*.

Le condizioni e le azioni possono avere tre contesti: *local*, ovvero cellula corrente, *neighbor*, ovvero cellule vicine, ed *environment*, ovvero ambiente extra-cellulare. Ogni contesto dovrà essere racchiuso tra parentesi quadre, e, alla fine delle diverse biomolecole coinvolte, dovrà essere indicato il tipo di

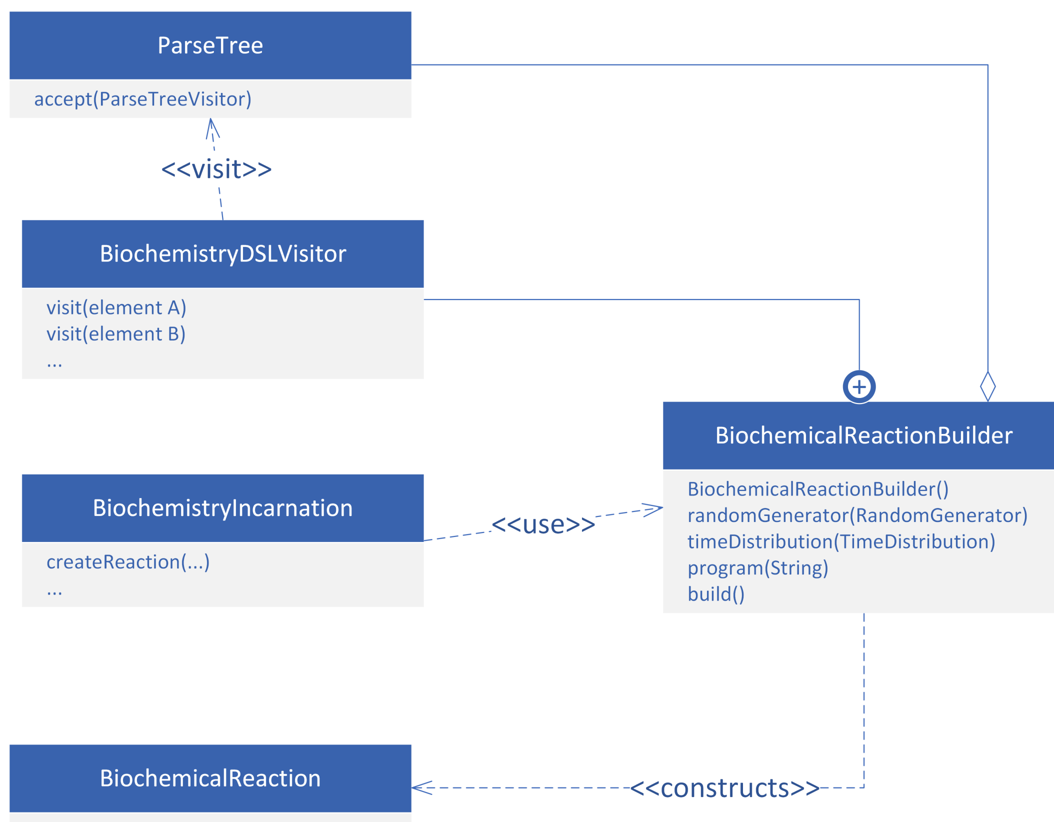
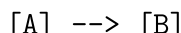
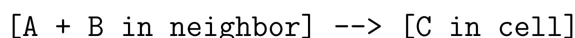
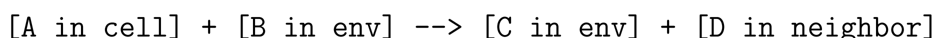


Figura 4.10: Diagramma delle classi UML che mostra la modellazione del pattern *builder* e del pattern *visitor*. Solo i metodi di interesse sono riportati, per non appesantire l'immagine. La classe `BiochemistryIncarnation`, si appoggia alla classe `BiochemicalReactionBuilder` per la creazione delle reazioni a partire da stringhe fornite dall'utente. Essa, come classe innestata, presenta la classe `BiochemistryDSLVisitor` (la freccia che collega le due classi è usata per modellare il concetto di classe innestata), che possiede molti metodi `visit`, uno per ogni elemento dell'albero sintattico. Esso, modellato dalla classe `ParseTree`, è contenuto nel builder, e formato dal parser del linguaggio, a partire dalla sequenza di token prodotti dal lexer. Se la stringa rappresentante la reazione è ben formattata, si produrrà, in uscita, una `BiochemicalReaction`.

contesto, tramite le parole chiave "`in cell`", "`in neighbor`" ed "`in env`". Se nessun contesto è indicato si assume automaticamente contesto cellulare. Le reazioni avranno quindi un aspetto del genere:





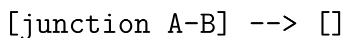


dove per indicare più azioni o condizioni si è usato il simbolo "`+`". Notare come ogni molecola deve essere all'interno di un contesto, pena il rilevamento di errori.

Per la modellazione di giunzioni si è usata una nuova keyword: "`junction`". Essa deve essere posizionata come prima parola all'interno di un contesto. Dopo di essa va scritto il nome della giunzione e subito chiuso il contesto. Non ci devono essere indicazioni sul tipo di contesto, o altre biomolecole. Solamente una ed una sola giunzione.

La creazioni di giunzioni, come detto, può avvenire solamente se è presente almeno una biomolecola nella cellula corrente, e una in un vicino. Inoltre il nome della giunzione deve riflettere il nome delle biomolecole coinvolte (vedi Sezione 4.1.4). La creazione di una giunzione avrà, quindi questa forma:

essa creerà la giunzione `A-B` nella cellula corrente, e quella `B-A` nel vicino scelto. La distruzione delle giunzioni avviene in modo implicito. Basterà includere la giunzione tra le condizioni, ma non tra le azioni:



distrugge la giunzione e riposiziona le molecole di cui era composta. Notare il contesto vuoto, che deve sempre essere presente nel caso non ci siano azioni o condizioni. Se la giunzione è presente sia a destra che a sinistra della reazione essa non viene distrutta:

questa reazione modella una *gap junction*, che, se presente, fa passare una molecola `A` dalla cellula corrente a quella collegata. Notare che in questo modo la giunzione comunicante è unidirezionale, poiché nella cellula vicina essa ha nome `B-A`. Sarà necessaria un'altra reazione per implementare anche l'altro lato della comunicazione.

Reazioni con condizioni o azioni custom

Non tutte le reazioni possono essere scritte in questo modo. Si pensi ad esempio alle reazioni che attuano il movimento cellulare, o a quelle che, tra le condizioni, hanno il numero delle cellule vicine. Questi concetti saranno modellati attraverso apposite classi Java, sottoclassi di *Action* o *Condition*, che saranno caricate, tramite *reflection*, dalle reazioni. Si è deciso di modellare le azioni custom come azioni che coinvolgono molecole. Esse sono posizionate in un contesto, da sole, senza alcuna indicazione. La sintassi dovrà essere simile a quella di un semplice costruttore Java: nome della classe e attributi fra parentesi tonde. Sia, ad esempio, la classe `BrownianMove` la classe java che implementa il moto browniano, e che come parametri del costruttore abbia un numero reale. Un esempio di reazione sarà:

```
[A in cell] --> [BrownianMove(0.1)]
```

La classe `BrownianMove` verrà caricata dinamicamente, tramite reflection, usando il costruttore che meglio si abbina ai parametri passati. Solo stringhe e numeri possono essere passati come parametri, tutte le altre astrazioni, come nodi, distribuzioni temporali o ambiente, sono passati in automatico alla classe, durante la creazione dell'istanza tramite reflection.

Per le azioni custom si è scelto un approccio simile. Le classi vengono caricate sempre tramite reflection, ma cambia il luogo della reazione dove devono essere scritte. Esse vanno posizionate dopo tutte le azioni e dopo la keyword "`if`". Se ci sono più condizioni custom si concatenano con il segno "`+`". Un esempio è:

```
[A] --> [A in neighbor] if NumberOfNeighborsGreaterThan(5)
```

4.5.2 Reazioni valide

Sono riportate di seguito un insieme di reazioni valide, secondo la grammatica del linguaggio progettato.

```
[] --> []
```

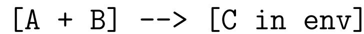
Reazione vuota, zero condizioni e zero azioni.

```
[A in cell] --> []
```

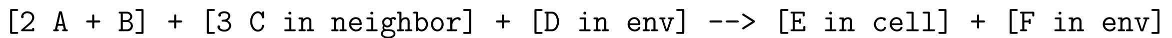
Distruzione, se presente, di una molecola `A` nella cellula.

```
[A] --> [B]
```

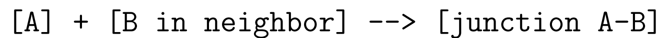
Trasformazione della molecola `A` in `B`. Se è presente elimina `A` dalla cellula, e aggiungi una molecola `B`



Se le molecole `A` e `B` sono presenti all'interno della cellula, eliminale e rilascia una molecola `C` nell'ambiente extra-cellulare.

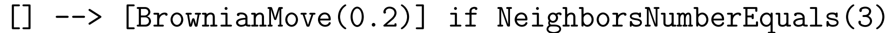


Se due molecole `A` e una `B` sono presenti all'interno della cellula, 3 molecole `C` sono contenute in una cellula adiacente, e una molecole `D` è dispersa nell'ambiente, crea una molecola `E` nella cellula e rilasci `F` nell'ambiente. Tutti i reagenti vengono eliminati.



Creazione di una giunzione, a partire dalla molecola `A` contenuta nella cellula, e `B` in un vicino.

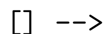
Modellazione di una *gap junction*. Se la cellula presenta una giunzione `A-B` ed una molecola `A`, passa `A` al vicino collegato con la giunzione.



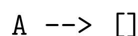
Attua il moto browniano della cellula, con parametro 0.2, se il numero di vicini è uguale a 3.

4.5.3 Reazioni non valide

A scopo di esempio, sono riportate alcune reazioni non valide, con la rispettiva spiegazione della loro non correttezza.



Manca il contesto (`[ ]`) delle azioni. Esso deve sempre essere presente, anche se vuoto.



Ogni condizione deve essere contenuta in un contesto.

La clausola giusta per il contesto *environment* è `in env`, non `in environment`

Non ci sono condizioni su molecole contenute in cellule vicine. Una giunzione, per essere formata, necessita di almeno una biomolecola nella cellula in cui è eseguita la reazione, ed una in una cellula adiacente.

```
[A in cell] + [B in neighbor] --> [junction X-Y]
```

Il nome delle giunzioni deve rispecchiare le molecole che la compongono. In questo caso sarebbe dovuto essere `A-B`

```
[A + B] + [C in neighbor] --> [junction A:2B-C]
```

Il nome della giunzione non rispecchia le molecole di cui è composta. Infatti, tra le condizioni, non c'è la presenza di due molecole `B`, ma solo di una.

```
[A + B] + [C in neighbor] --> [junction A:B-C + D in env]
```

Il contesto che contiene una giunzione, può contenere solamente una ed una sola giunzione. Altre molecole devono essere posizionate in contesti diversi. La reazione dovrebbe aver forma: `[junction A:B-C] + [D in env]`.

```
[junction A-B] + [A] --> [junction X-Y]
```

Le due giunzioni presenti, una tra le azioni ed una tra le condizioni non sono dello stesso tipo. C'è ambiguità tra l'uso della giunzione `A-B` o la creazione della giunzione `X-Y`

```
[A + B] --> CustomAction(1.0)
```

Le azioni devono essere contenute in un contesto, anche se sono custom. La sintassi corretta, per l'azione, sarebbe `[CustomAction(1.0)]`

```
[CustomCondition(3)] --> [BrownianMove(0.2)]
```

Le condizioni custom vanno posizionate dopo tutte le azioni, e dopo la parola chiave `if`.

Capitolo 5

Implementazione

In questo capitolo vengono trattati i dettagli implementativi più rilevanti dell'applicazione sviluppata.

Il simulatore Alchemist, e quindi anche l'incarnazione Biochemistry, sono sviluppati nel linguaggio di programmazione Java, facendo largo uso delle novità, introdotte con l'ultima release: Java 8. Oltre alle librerie standard di Java, sono state usate alcune librerie esterne, come Guava[1].

Lo sviluppo dell'incarnazione Biochemistry è avvenuto mantenendo gli standard qualitativi del codice di Alchemist. Si è sviluppato attraverso l'IDE Eclipse[2], con gli strumenti di qualità del codice *CheckStyle*, *PMD* e *FindBugs* attivi, e con la configurazione consigliata dai maintainer del simulatore.

Il codice è pubblicamente accessibile su *github*, al seguente indirizzo web: `https://github.com/ggraffieti/Alchemist/tree/feature-biochemistry`. La continuous integration del software è stata attuata sia tramite la build automatica con gradle, sia attraverso il sito web `drone.io`. Esso, ogni volta che viene fatto un cambiamento al progetto, compie una build completa, più l'esecuzione di tutti i test su una macchina virtuale pulita. Ciò permette di prevenire la propagazione degli errori, potendoli scoprire immediatamente. Il report delle build dell'incarnazione Biochemistry può essere consultato al seguente link: `https://drone.io/github.com/ggraffieti/Alchemist`.

5.1 Linguaggio Biochemistry

Come decritto in fase di design, si è scelto di modellare, il linguaggio per la scrittura di reazioni, come un piccolo DSL. Si è progettato un insieme di classi per la sua interpretazione, ognuna con un compito specifico: *lexer, parser*

[1] `https://github.com/google/guava`

[2] `https://eclipse.org`

e visitor. La scrittura di lexer e parser, tuttavia, non è un compito facile. Al giorno d'oggi, per piccoli linguaggi, essi sono generati automaticamente da diversi tool, che li producono a partire da una grammatica. La specifica della grammatica per un linguaggio dipende, quindi, dallo strumento utilizzato per la generazione del parser. Il visitor, al contrario, è stato implementato da zero, poiché esso, visitando il *syntax tree*, deve produrre le classi proprie dell'incarnazione Biochemistry.

Come detto, esistono diversi strumenti per la generazione di parser, a partire da una ben definita grammatica. Tra i più famosi, per il linguaggio Java, ci sono JavaCC (*Java Compiler Compiler*), Xtext, e ANTLR. Dei tre il più completo risulta sicuramente Xtext. Esso, oltre e generare parser e lexer per il linguaggio, genera in automatico anche diversi tool integrati con Eclipse, come il *syntax coloring* o la *code completion*.

5.1.1 Strumenti utilizzati

Inizialmente è stato scelto Xtext per l'implementazione del DSL. Tuttavia, valutando meglio la dimensione del progetto, si è scelto di optare per ANTLR, che offre un servizio più minimale: vengono generati, infatti, solamente il parser ed il lexer. ANTRL (*ANother Tool for Language Recognition*)[3], è un generatore di parser di tipo LL(*), ovvero parsa l'imput da sinistra verso destra (*Left to right*) ed esegue una derivazione da sinistra (*Leftmost derivation*) della grammatica. ANTRL produce quindi un parser *top-down*, che parte dal contesto più ampio, fino ad arrivare ai token terminali.

Plug-in gradle

La generazione di lexer e parser a partire dalla grammatica dovrà avvenire in modo automatico. Solo quest'ultima, infatti, dovrà essere inclusa nella distribuzione dell'incarnazione Biochemistry. Solo durante la build saranno generate, da ANTLR, le classi Java del parser e del lexer. Per implementare ciò si è fatto uso del plug-in ANTLR per gradle. Esso aggiunge un *task* a quelli già presenti in Alchemist, chiamato `generateGrammarSource`, che, come suggerisce il nome, genera le classi Java partendo dalla grammatica. Essa deve essere inserita, in un file con estensione "`.g4`" nella source folder `src/main/antlr`. Il task `generateGrammarSource` aggiunge una dipendenza al task che compila le classi java. Quest'ultimo può essere eseguito solo dopo la generazione della grammatica. Un piccolo esempio del codice usato per la regolazione del plug-in è visibile in Codice 5.1. Esso è inserito nel file `build.gradle` dell'incarnazione Biochemistry.

[3] `http://www.antlr.org`

```
apply plugin: 'antlr'

generateGrammarSource {
  arguments += ["-visitor", "-package",
      "it.unibo.alchemist.biochemistrydsl"]
  outputDirectory = new File('src/antlr/java')
}
compileAntlrJava.dependsOn('generateGrammarSource')
```

Codice 5.1: Plug-in ANTLR per gradle. Il task `generateGrammarSource` prende in ingresso, come argomenti, il parametro `-visitor`, il quale consente di produrre anche una classe astratta da prendere come base per la costruzione del visitor, e `-package`, il quale inserisce, come prima riga delle classi generate, l'indicazione del package di appartenenza. La riga 5 consente di specificare in quale cartella saranno generate le classi. Nella riga 7 si fa dipendere la compilazione delle classi, prodotte da ANTLR, dalla loro generazione.

5.1.2 Grammatica

ANTLR usa una sintassi molto simile alla *Extended Backus–Naur Form* (EBNF) per la scrittura della grammatica. Le regole definite da essa si dividono in due grandi famiglie: le regole lessicali (*lexical rules*), che rappresentano i lessemi del linguaggio, come ad esempio una stringa o un numero intero. Anche le keyword del linguaggio sono considerate regole lessicali. L'altra famiglia sono le regole grammaticali (*grammar rules*), che indicano come combinare i lessemi per formare programmi validi. Le regole lessicali possono essere considerate come le parole della lingua italiana, mentre le regole grammaticali indicano l'ordine in cui devono essere scritte le parole, per formare frasi di senso compiuto. Le regole grammaticali sono organizzate secondo una struttura ad albero, dove le regole più generali fanno uso di regole più specifiche. L'ordine con cui sono specificate le regole è importante, poiché determina l'ordine in cui esse sono valutate. Si deve, quindi, sempre partire dalle regole più generali, per poi arrivare a quelle più specifiche. ANTRL consente, inoltre, la scrittura di regole ricorsive.

Per convenzione, una regola lessicale ha nome tutto maiuscolo, mentre una regola grammaticale minuscolo. Nella grammatica implementata ci sono solamente tre regole lessicali: `LITERAL`, che rappresenta una stringa, la quale inizia con un carattere alfabetico, ed è formata solo da lettere, numeri ed underscore, `POSDOUBLE`, che rappresenta un numero reale positivo, e `WS`, il quale modella un qualsiasi carattere di whitespace. Esso viene saltato durante il parsing, per cui gli spazi o i tab non sono importanti ai fini della corretta interpretazio-

ne del linguaggio. Soltanto con queste poche regole lessicali è stato possibile la scrittura di una grammatica per il DSL modellato nella Sezione 4.5. Un esempio di grammatica, usata per la modellazione delle giunzioni nel linguaggio Biochemistry, è presente in Codice 5.2. Tutta la grammatica è consultabile al seguente link: `https://github.com/ggraffieti/Alchemist/blob/feature-biochemistry/alchemist/alchemist-incarnation-biochemistry/src/main/antlr/it/unibo/alchemist/biochemistrydsl/Biochemistrydsl.g4`.

```
junction
  : 'junction' junctionLeft '-' junctionRight
  ;

junctionLeft
  : biomolecule (':' biomolecule)*
  ;

junctionRight
  : biomolecule (':' biomolecule)*
  ;

biomolecule
  : (concentration)? name=LITERAL
  ;

concentration
  : POSDOUBLE
  ;
```

Codice 5.2: Definizione di giunzione, nel DSL Biochemistry. Nella riga 1 è definita la regola grammaticale che modella una giunzione. Essa è formata dalla keyword `junction` (in questo caso non inclusa tra le regole lessicali, poiché presente solo in questa porzione di codice), dalla regola `junctionLeft`, il segno "-" e la regola `junctionRight`. Esse sono formate obbligatoriamente da almeno una biomolecola, ma possono contenerne anche più di una, separate dal simbolo ":" (riga 6 e 10). Anche se uguali si sono fatte due regole distinte, poiché, in fase di visita del syntax tree, è necessario conoscere quali molecole sono a sinistra, e quali a destra nel nome della giunzione. La regola `biomolecule` è formata da una concentrazione (opzionale), e un nome, modellato attraverso la regola lessicale `LITERAL`. La concentrazione, a sua volta, è formata dalla regola `POSDOUBLE`. Notare come sia mantenuta la gerarchia per le regole grammaticali: dalla più generale alla più specifica.

Le regole lessicali sono, al contrario, formate solo da espressioni regolari (*regex*). Le regole lessicali del linguaggio Biochemistry sono visibili in Codice 5.3.

```
POSDOUBLE
  : [0-9]+ ('.' [0-9]*)?
  ;

LITERAL
  : ([a-zA-Z]) ([a-zA-Z0-9] | '_')*
  ;

WS
  : [ \t\r\n]+ -> skip
  ;
```

Codice 5.3: Regole sintattiche del DSL Biochemistry. Notare, nella riga 10 l'indicazione `-> skip`. Ciò significa che quella regola lessicale non deve essere valutata in fase di parsing.

5.1.3 Visitor

Dopo il processo di parsing si ottiene un *syntax tree*, formato da tutte le regole grammaticali trovate, ordinato secondo una struttura gerarchica. Per formare le reazioni specifiche per l'incarnazione Biochemistry, si deve visitare il suddetto albero. ANTRL fornisce una classe astratta, chiamata `BiochemistrydslBaseVisitor`, la quale presenta un'implementazione di default dei metodi atti a visitare le regola contenute nell'albero sintattico. Essi hanno la forma: `visit<Nome regola>` e prendono in ingresso il contesto della regola da visitare. Il contesto, o *context*, di una regola, non è altro che il sotto albero formato da tutti i nodi del *syntax tree* discendenti della regola visitata. La classe astratta in esame è parametrica. Ogni metodo ha come valore di ritorno un valore generico `T`.

Si è estesa la classe astratta descritta sopra, tramite la classe `BiochemistryDSLVisitor`. Essa è una classe innestata statica, contenuta nella classe builder delle reazioni. Come valore di ritorno dei metodi si è scelta la reazione biochimica da costruire, modellata dalla classe `BiochemicalReaction`. Non tutti i metodi della superclasse sono stati implementati. L'albero viene visitato fino a che non è chiara l'azione o la condizione da aggiungere alla reazione. Una volta scoperta, tramite metodi privati, si estraggono dal contesto della regola le informazioni necessarie, e si aggiunge, alla reazione da costruire, la condizione

o azione creata. Un esempio di questo comportamento è visibile in Codice 5.4 e Codice 5.5.

```
@Override
public Reaction<Double>
    visitBiochemicalReactionLeftInCellContext(final
    BiochemicalReactionLeftInCellContextContext ctx) {
  for (final BiomoleculeContext b : ctx.biomolecule()) {
    final Biomolecule biomol = createBiomolecule(b);
    final double concentration = createConcentration(b);
    insertInMap(biomolConditionsInCell, biomol, concentration);
    conditionList.add(new BiomolPresentInCell(biomol, concentration,
        (CellNode) node));
    actionList.add(new ChangeBiomolConcentrationInCell(biomol,
        -concentration, (CellNode) node));
    }
  return reaction;
}
```

Codice 5.4: Visita della regola che modella il contesto delle condizioni cellulari. Dal contesto si ricavano le biomolecole presenti (riga 3), e si compie un ciclo su di loro. Vengono quindi create la molecole e la rispettiva concentrazione, attraverso metodi privati, che, preso in ingresso il contesto di una biomolecola, restituiscono i valori voluti. Nelle righe 7 e 8 vengono aggiunte le condizioni di presenza della molecola, con il valore di concentrazione trovato, e l'azione che la elimina (le molecole reagenti vengono eliminate dalla reazione). La reazione viene poi ritornata dal metodo. La riga 6 verrà discussa di seguito.

```
private static Biomolecule createBiomolecule(final
    BiomoleculeContext ctx) {
  return new Biomolecule(ctx.name.getText());
}
```

Codice 5.5: Classe privata, contenuta nel visitor, che crea una molecola a partire dal suo contesto.

La creazione di condizioni o azioni custom avviene tramite reflection. Le uniche cose note, della classe da caricare, sono il nome e la lista dei parametri. Se esiste una classe con il nome dato in input, si procede a provare tutti i costruttori, passando in ingresso i parametri, nell'ordine dato. Se ci sono parametri impossibili da conoscere in fase di scrittura della reazione, come il nodo o l'ambiente, essi sono istanziato automaticamente. Se nessun costruttore si adatta ai parametri dati in input viene mostrato un messaggio di errore.

Precedentemente si è detto come, nella fase di parsing, vengono rilevati eventuali errori nella reazioni data in ingresso. Solo quelli sintattici e semantici possono però essere intercettati. Come si è trattato nel Capitolo 3, una giunzione deve avere un nome il quale riflette la sua composizione: molecole coinvolte nella cellula corrente "-" molecole coinvolte nella cellula vicina. Il controllo su questa regola non può essere fatta a livello di parsing, poiché non può essere espressa dalla grammatica. Deve essere quindi intercettata durante la visita dell'albero sintattico.

Durante la visita delle condizioni cellulari e sui vicini, le molecole coinvolte vengono inserite in una mappa, con il rispettivo valore di concentrazione. Se si visita la regola che definisce la creazione di una giunzione, viene fatto un controllo sulle molecole di cui è composta. Se una biomolecola non è presente nella mappa, o è presente con una concentrazione minore rispetto a quella usata per la giunzione, essa non può essere creata. Viene quindi stampato un messaggio di errore e stoppato il caricamento della simulazione. Questo comportamento è visibile in Codice 5.6.

```
1  @Override
2  public Reaction<Double> visitCreateJunctionJunction(final
       BiochemistrydslParser.CreateJunctionJunctionContext ctx) {
3    final Junction j = createJunction(ctx.junction());
4    j.getMoleculesInCurrentNode().forEach((k, v) -> {
5      if (!biomolConditionsInCell.containsKey(k) ||
           biomolConditionsInCell.get(k) < v) {
6        throw new BiochemistryParseException("The creation of the
             junction " + j + " requires " + v + " " + k + " in the
             current node, specify a greater or equal value in
             conditions.");
7      }
8    });
9    j.getMoleculesInNeighborNode().forEach((k, v) -> {
10     if (!biomolConditionsInNeighbor.containsKey(k) ||
           biomolConditionsInNeighbor.get(k) < v) {
11         throw new BiochemistryParseException("The creation of the
               junction " + j + " requires " + v + " " + k + " in the
               neighbor node, specify a greater or equal value in
               conditions.");
12     }
13   });
14   actionList.add(new AddJunctionInCell(j, node, env, rand));
15   actionList.add(new AddJunctionInNeighbor(reverseJunction(j), node,
         env, rand));
16   return reaction;
```

```
17 }
```

Codice 5.6: Creazione dell'azione di costruzione di una giunzione. La giunzione viene creata nella riga 3. Nelle righe successive vengono controllate le molecole da cui essa è formata. Se tra le condizioni non ci sono abbastanza biomolecole di quel tipo (sia che siano contenute nella cellula in esame, sia nella cellula vicina), viene mandato a video un messaggio di errore, attraverso la generazione di una `BiochemistryParseExeption`, che termina anche il processo di caricamento della simulazione. Se invece tutto va a buon fine, vengono aggiunte alla reazione le azioni che creano la giunzione, sia nella cellula corrente che in quella vicina.

5.2 Incarnazione Biochemistry

La progettazione dell'incarnazione Biochemistry è stata ampiamente trattata nel Capitolo 4. In questa sezione verranno esposte le tecniche implementative più interessanti. Tutto il codice prodotto durante questo progetto è visibile all'indirizzo web : `https://github.com/ggraffieti/Alchemist/tree/feature-biochemistry/alchemist/alchemist-incarnation-biochemistry`.

5.2.1 Nodi cellulari

I nodi cellulari devono contenere al loro interno molecole, giunzioni e reazioni. Queste ultime sono inserite in una lista, mentre le molecole in una mappa, che ha come chiave la molecola stessa, e come valore la rispettiva concentrazione.

Le giunzioni meritano una trattazione più approfondita. Un tipo di giunzione, infatti, può collegare la cellula a più nodi diversi, e lo stesso nodo può essere collegato, con il medesimo tipo di giunzione, più volte. Si è deciso di implementare questi legami tramite una mappa di mappe. La mappa più esterna ha come chiave il tipo di giunzione, e, come valore, una ulteriore mappa, con chiave il nodo collegato, e valore il numero di giunzioni di quello specifico tipo con quel nodo. Essa ha la forma di `Map<Junction, Map<ICellNode, Integer>>`. Questo approccio consente un accesso ottimale, sia in lettura che scrittura. Ad esempio, dato un tipo di giunzione, conoscere quali nodi sono collegati alla cellula corrente, con la giunzione data, richiede un tempo di accesso molto basso. Questa struttura dati è implementata tramite il `MapMaker` presente nella libreria *guava.* Esso consente di creare mappe con livello di concorrenza scelto dal progettista. Ciò permette, ad un numero fissato di thread,

la modifica concorrente della mappa. Come *concurrency level* si è scelto il valore 2.

La concentrazione delle molecole è un numero reale, e deve sempre essere maggiore di zero. Se esso raggiunge il valore zero, la molecola deve essere eliminata dalla mappa. Questo è stato implementato tramite *override* del metodo `setConcentration`. È stato, inoltre, modificato il metodo `contains(Molecule)`. Dato che una giunzione è modellata come un particolare tipo di molecola, se quella passata è un'istanza di giunzione esso ritorna il numero totale di giunzioni del tipo specificato, presenti nella cellula.

Il metodo `removeJunction(Junction, Node)`, oltre ad occuparsi della rimozione della giunzione, del tipo specificato, che collega la cellula al nodo vicino passato in input, si occupa del reinserimento delle molecole nella cellula.

5.2.2 Reazioni

Le reazioni biochimiche, usate dall'incarnazione Biochemistry, sono state implementate nella classe `BiochemicalReaction`. Esse vengono create dalla classe `BiochemicalReactionBuilder`, come discusso nelle sezioni precedenti. La reazione viene creata nel metodo `build()`, facendo uso delle strutture dati per la gestione del DSL, come mostrato in Codice 5.7.

```
1 public Reaction<Double> build() {
2   checkReaction();
3   final BiochemistrydslLexer lexer = new BiochemistrydslLexer(new
       ANTLRInputStream(reactionString));
4   final BiochemistrydslParser parser = new BiochemistrydslParser(new
       CommonTokenStream(lexer));
5   parser.removeErrorListeners();
6   parser.addErrorListener(new
       BiochemistryParseErrorListener(reactionString));
7   final ParseTree tree = parser.reaction();
8   final BiochemistryDSLVisitor eval = new
       BiochemistryDSLVisitor(rand, incarnation, time, node, env);
9   return eval.visit(tree); }
```

Codice 5.7: Creazione di una `BiochemicalReaction`, a partire da una stringa, scritta secondo le regole del DSL Biochemistry. Nella riga 2 viene controllato che tutti i valori passati al builder siano corretti. Si crea quindi il lexer, che performa l'analisi lessicale della reazione. Si costruisce il parser passandogli in ingresso lo stream di token prodotto dal lexer. Si aggiunge un listener custom per gli errori di pasing, e si genera l'albero sintattico (riga 7). Si crea quindi il visitor del suddetto albero, e si compie la visita, producendo la rezione desiderata.

Il calcolo del rate della reazione è fatto moltiplicando tra loro tutte le *propensity function* delle condizioni, ed il rate statico della reazione. Una condizione su molecole calcola la sua propensità come: $\binom{c}{r}$, dove c rappresenta la concentrazione della molecola, e r il numero minimo di molecole richieste. Per il calcolo del coefficiente binomiale si è fatto uso della libreria *common math* di Apache[4]. Le condizioni che non comprendono molecole chimiche hanno una propensità che varia da caso a caso. Ad esempio, la presenza di una giunzione cellulare ha propensità pari a uno, se essa è presente.

Per il corretto funzionamento del grafo delle dipendenze tra reazioni, ogni condizione deve rendere note le molecole che saranno lette. Allo stesso modo, un'azione deve registrare le molecole modificate. Questo avviene attraverso i metodi `addReadMolecule` e `addModifiedMolecule`, che, rispettivamente, inseriscono le molecole tra quelle lette e quelle modificate dalla reazione. Questi metodi sono chiamati nei costruttori delle condizioni e delle azioni. Nel caso di rottura di una giunzione cellulare, devono essere registrate come molecole modificate, tutte quelle facente parti della giunzione, che saranno reimmesse nella cellula.

5.2.3 Reazioni multi-cellulari

Le reazioni con il vicinato sono un particolare tipo di reazioni biochimiche, le quali hanno necessità di una maggiore attenzione in fase implementativa, rispetto alle reazioni intracellulari. Il modello di questo tipo di reazioni è stato discusso nella Sezione 4.2. L'aspetto più importante è, senza dubbio, l'unione delle condizioni: tutte le condizioni su cellule vicine devono essere soddisfatte dalla stessa cellula. Si è aggiunto un metodo, nelle condizioni, chiamato `getValidNeighbors`, il quale, ricevuta in ingresso una lista di nodi, esegue un filtraggio in base alla condizione da valutare, e ritorna solo i nodi validi, a partire da quelli dati in ingresso. L'implementazione di questo metodo è visibile in Codice 5.8.

La reazione biochimica, modellata in `BiochemicalReaction`, dovrà modellare l'unione delle condizioni attraverso il metodo `updateInternalStatus`. Esso viene richiamato ogni qualvolta la reazione è eseguita, o è stata computata una delle sue dipendenze. Esso, se sono presenti reazioni sui vicini dovrà testarle tutte, passando alla prima la lista dei vicini, ricevendo come valore di ritorno la lista dei vicini che soddisfano quella condizione. Passando come parametro quest'ultima lista alla eventuale seconda reazione sul vicinato, si ottengono i nodi adiacenti che soddisfano entrambe le condizioni. Procedendo in questo modo per tutte le condizioni sul vicinato si ottiene la lista delle cellule adiacenti che soddisfano tutte le condizioni. Come si è visto dal Codice 5.8,

[4] `http://commons.apache.org/proper/commons-math`

la condizione restituisce anche il rispettivo valore della propensità di ogni nodo. Esso dovrà essere aggregato, per tutte le condizioni, in modo da ricavare la propensità totale per ogni cellula. Il metodo `updateInternalStatus` della reazione biochimica è visibile in Codice 5.9.

```java
@Override
public Map<Node<Double>, Double> getValidNeighbors(final
    Collection<? extends Node<Double>> neighborhood) {
  propensity = 0;
  neigh = neighborhood.stream()
    .filter(n -> n.getConcentration(mol) >= conc)
    .collect(Collectors.<Node<Double>, Node<Double>, Double>toMap(
      n -> n,
      n -> CombinatoricsUtils.binomialCoefficientDouble(
        n.getConcentration(mol).intValue(), conc.intValue())));
  if (!neigh.isEmpty()) {
    propensity = neigh.values().stream().max((d1, d2) ->
        d1.compareTo(d2)).get();
  }
  return new HashMap<>(neigh);
}
```

Codice 5.8: Metodo `getValidNeighbors` della condizione sulla presenza di una biomolecola in un vicino, con un valore di concentrazione dato. Si compie uno `stream` sulla collezione di nodi passata in ingresso, e si filtrano solo quelli che soddifano la condizione. Dopo di che si aggregano in una mappa, che ha come chiave il nodo che soddisfa la condizione, e come valore il suo valore di propensità, calcolato come coefficiente binomiale di concentrazione su molecole richieste. Se c'è almeno una cellula che soddisfa la condizione si calcola la propensità della condizione, usando il valore massimo tra quelli delle cellule filtrate. Si restitisce poi la mappa. Notare che non è stato fatto alcun controllo sul fatto che i nodi passati come parametro siano effettivamente vicini. Sarà compito della reazione assicurarsi la correttezza di ques'assunzione.

Se sono presenti anche azioni sul vicinato, il nodo sul quale eseguirle (bersaglio), viene scelto come quello con la propensità più alta tra i nodi validi. Se non ci sono condizioni sul vicinato, ma solamente azioni su di esso, il nodo dove eseguirle viene, invece, scelto in modo casuale. Ovviamente le azioni sul vicinato possono essere svolte soltanto se il nodo in esame presenta almeno un vicino. Se non ci sono condizioni sui vicini viene aggiunta, automaticamente in fase di creazione della reazione, una ulteriore condizione, atta a verificare la presenza di almeno una cellula adiacente. Ciò è necessario poiché se non ci sono condizioni sul vicinato, ma solo azioni su di esso, la reazione veniva

schedulata anche se la cellula era isolata.

```
@Override
protected void updateInternalStatus(final Time curTime, final
    boolean executed, final Environment<Double> env) {
  if (neighborConditionsPresent) {
    validNeighbors.clear();
    validNeighbors = env.getNeighborhood(node).getNeighbors()
      .stream()
      .collect(Collectors.<Node<Double>, Node<Double>, Double>toMap(
        n -> n,
        n -> 0d));
    for (final Condition<Double> cond : getConditions()) {
      if (cond instanceof AbstractNeighborCondition) {
        validNeighbors = intersectMap(validNeighbors,
            ((AbstractNeighborCondition<Double>)
            cond).getValidNeighbors(validNeighbors.keySet()));
        if (validNeighbors.isEmpty()) {
          break;
        }
      }
    }
  }
  super.updateInternalStatus(curTime, executed, env);
}
```

Codice 5.9: Override del metodo `updateInternalStatus` nella classe `BiochemicalReaction`. Se sono presenti condizioni sui vicini (riga 3) ricalcola i vicini validi, partendo dalla lista di tutti i nodi nel vicinato. Per ognuna delle suddette condizioni calcola l'insieme dei nodi validi, partendo da quelli validi per le condizioni precedenti. Il metodo `intersectMap` interseca la mappa dei nodi validi, con il sottoinsieme di essi validi anche per la condizione in esame, sommandone la propensità. Se i nodi validi sono zero si esce dal ciclo. Nella riga 14 si richiama il metodo implementato nella superclasse. Esso si occupa di calcolare il rate totale della reazione partendo da quello delle condizioni. Se non sono presenti condizioni sul vicinato viene eseguito soltanto il metodo della classe padre.

5.2.4 Giunzioni

Le giunzioni sono state implementate attraverso la classe `Junction`, la quale implementa l'interfaccia *Molecule*. Esse, oltre al nome, sono caratterizzate da due mappe: una contenente le molecole, formanti la giunzione, che erano

contenute nel nodo in cui l'istanza di giunzione è contenuta, la seconda le molecole del nodo collegato. Entrambe, come valore, presentano la concentrazione delle molecole.

Come visto in fase di trattazione del movimento cellulare, se un nodo, muovendosi, si trova nella situazione di avere giunzioni con cellule non più presenti nel suo vicinato, esse devono, immediatamente, essere distrutte. Dato che il movimento dei nodi è gestito dall'ambiente, si è prodotta una sua estensione, chiamata `BioRect2DEnvironment`. Essa, tramite il metodo `moveNode`, gestisce la corretta funzionalità descritta sopra. L'implementazione del metodo è riportata in Codice 5.10. Il metodo `removeJunction`, oltre a rimuovere la giunzione, reimmette le molecole, da cui era formata la giunzione, nella cellula di origine.

```
1  @Override
2  public void moveNode(final Node<Double> node, final Position
       direction) {
3    if (node instanceof CellNode) {
4      super.moveNode(node, direction);
5      final Neighborhood<Double> neigh = getNeighborhood(node);
6      final Map<Junction, Map<ICellNode, Integer>> jun = ((CellNode)
          node).getJunctions();
7      jun.entrySet().stream().forEach(e ->
          e.getValue().entrySet().forEach(e2 -> {
8        if (!neigh.contains(e2.getKey())) {
9          for (int i = 0; i < e2.getValue(); i++) {
10           ((CellNode) node).removeJunction(e.getKey(), e2.getKey());
11           e2.getKey().removeJunction(e.getKey().reverse(),
                 (ICellNode) node);
12         }
13       }
14     }));
15   }
16 }
```

Codice 5.10: Rimozione delle giunzioni, non più nel vicinato, dopo il movimento della cellula. In riga 5 si ottiene il nuovo vicinato, si iterano tutte le giunzioni del nodo, per ogni nodo collegato. Se ne esiste uno collegato, ma non più nel vicinato, le corrispettive giunzioni vengono eliminate.

Capitolo 6

Test

In questo capitolo vengono esposti i test compiuti sull'incarnazione Biochemistry sviluppata, e si commentano brevemente i risultati ottenuti.

Lo sviluppo del progetto ha seguito il modello del *test driven development* (TDD), dove i test vengono scritti prima dell'implementazione vera e propria dell'applicazione. Ciò permette di avere un'idea più chiara degli obiettivi del progetto, senza perdersi in inutili dettagli implementativi, superflui nelle prime fasi di sviluppo. Soltanto quando si produce una prima implementazione funzionante si passa alla fase di *refactoring* del codice, dove vengono attuate tutte le ottimizzazioni necessarie, mantenendo come obiettivo il passaggio del test.

Per poter attuare uno sviluppo *test driven*, è necessaria la generazione di test automatici, almeno sulle parti centrali del progetto. Essi dovranno essere eseguiti ogni qualvolta viene fatto un cambiamento, e devono sempre dare esito positivo. Come visto, nei capitoli precedenti, si è fatto uso di `drone.io` per l'esecuzione automatica, sia della build del progetto, sia dei test automatici. Esso è collegato con il progetto di Alchemist, ospitato su github. Ogni qualvolta viene rilevato un cambiamento, esso fa partire una build completa (utilizzando gradle), su una macchina virtuale Ubuntu Linux completamente pulita. I risultati delle varie build sono notificati tramite email, per cui l'eventuale fallimento dei test è rilevato immediatamente.

Si sono scritti vari test automatici, sotto forma di *unit test*, per le varie parti del sistema. Essi sono stati implementati attraverso il framework *JUnit*, usato per la generazione di test per il linguaggio Java. Essi comprendono:

- Un test per il DSL Biochemistry, in cui, dato un insieme di reazioni valide, ed uno con errori semantici o sintattici, si verifica il corretto parsing delle stringhe, e la restituzione di eccezioni nel caso di errori.

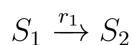
- Un test del caricamento delle reazioni nel sistema. Mentre quello precedente verificava solamente il corretto parsing del linguaggio, in base alla grammatica, questo test verifica il corretto funzionamento della sua interpretazione. Esso verifica il caricamento delle condizioni e delle azioni giuste, in base alle regole implementate nel visitor per il DSL.
- Un test sulla creazione e il confronto tra molecole chimiche.
- Un test sulle giunzioni cellulari. Esso comprende la loro creazione e distruzione, in modo da verificarne il corretto funzionamento.

Tutti i test automatizzati sono consultabili nella source folder `src/test` del progetto.

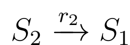
6.1 Test intracellulare

Oltre ai già citati test automatici, sono state compiute diverse simulazioni di prova, usando sistemi noti in letteratura, e confrontando i risultati ottenuti con quelli divulgati. Anche in questo caso i test sono stati divisi per categorie: intracellulari, comunicazione, giunzioni e sistemi reali.

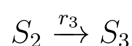
Il primo test compiuto per verificare la correttezza dell'applicazione, nella sua estensione prettamente intracellulare, è stata la riproduzione di una simulazione, portata come esempio da Gillespie in [11]. Essa consta di sole tre molecole, e tre reazioni che ne modellano la dinamica. Esse sono riportate di seguito:



$$S_1 \xrightarrow{r_1} S_2$$

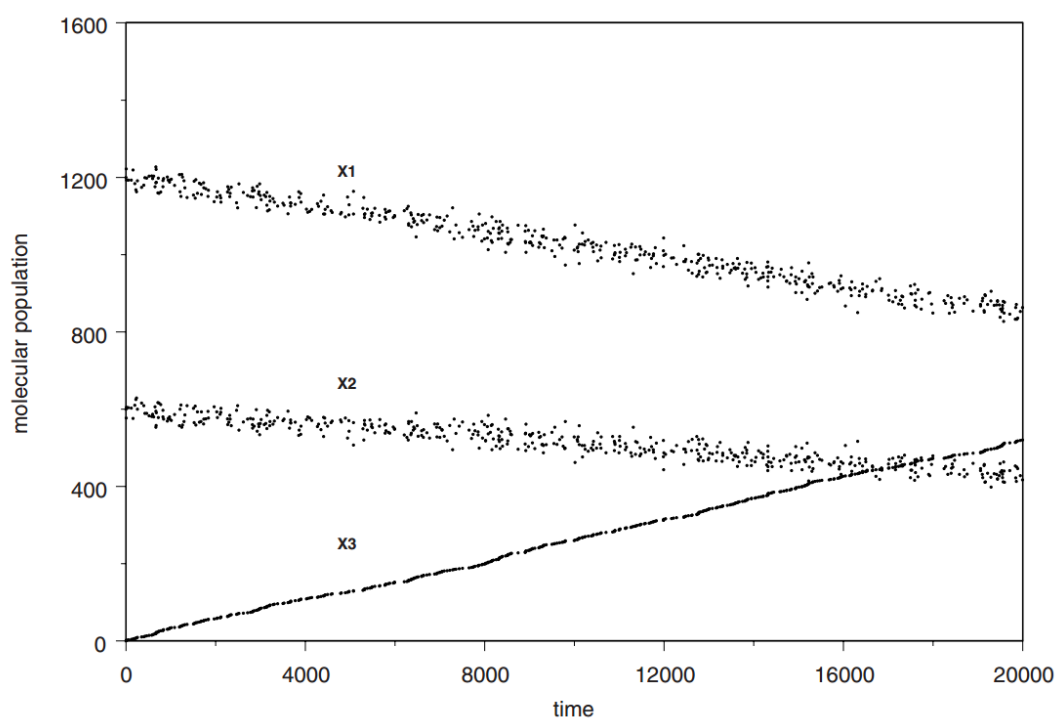


$$S_2 \xrightarrow{r_2} S_1$$

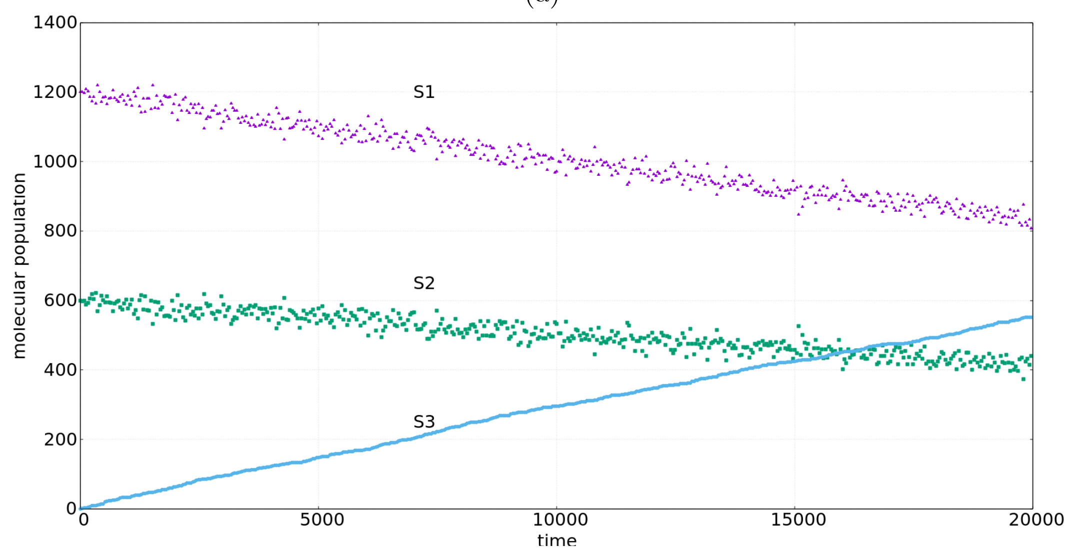


$$S_2 \xrightarrow{r_3} S_3$$

Con $r_1 = 1$, $r_2 = 2$ ed $r_3 = 5 \times 10^{-5}$. Condizioni iniziali: $[S_1] = 1200$ molecole, $[S_2] = 600$ molecole e $[S_3] = 0$ molecole. Dato il basso rate della terza reazione, è stato necessario un tempo di simulazione piuttosto lungo, pari a 20 000 *time unit*. In questo intervallo temporale, sono state eseguite circa 40 milioni reazioni. Il test è stato eseguito su una singola cellula, compiendo 30 simulazioni con altrettanti seed diversi tra loro. I risultati sono visibili in Figura 6.1.



(a)



(b)

Figura 6.1: Raffronto tra i grafici ottenuti per la reazione descritta sopra, ed usata come test per la dinamica intracellulare. Il grafico in (a) è ottenuto dall'articolo di Gillespie [11] usato come riferimento, mentre il grafico in (b) è stato prodotto a partire da dati raccolti durante una simulazione del sistema.

6.2 Test della comunicazione

Procedendo nello stesso modo in cui si è modellata e sviluppata l'applicazione, si trattano ora i test della comunicazione, tra cellule adiacenti. Un primo, ed importante requisito della comunicazione, è il mantenimento delle molecole scambiate. Esse, infatti, non devono venire perse o create dal nulla durante la comunicazione. Come si è discusso nella Sezione 4.4.2, il movimento cellulare, unito alla comunicazione poteva portare a inconsistenza nelle simulazioni, compresi problemi di generazione dal nulla di molecole chimiche.

Per verificare la corretta modellazione delle dinamiche multi-cellulari, si è scritta una simulazione, il cui unico scopo è verificare che il numero delle molecole rimanga costante, anche a fronte di comunicazione e movimento. Essa è formata da una griglia di cellule, di dimensioni variabili in ogni simulazione, ed una cellula centrale, contenente 1 000 molecole `A`. Le uniche azioni implementate sono le seguenti (scritte secondo il DSL Biochemistry):

```
[A in neighbor] --> [A]
```

oppure

```
[A] --> [A in neighbor]
```

e

```
[] --> [BrownianMove(0.8)]
```

con rate variabili da simulazione a simulazione. Ogni parametro (seed della simulazione, dimensione della grigli e rate delle due reazioni), varia sulla base di sei valori distinti. È stata compiuta una simulazione per ogni diversa combinazione di parametri, quindi un totale di $6^4 = 1296$ simulazioni. Ognuna di esse simula un intervallo temporale pari a 1000 *time unit*. Si sono aggregati i dati e tutte le simulazioni mostravano un numero di molecole costante, pari alle condizioni iniziali.

Il secondo test, sulla comunicazione cellulare, mira a simulare il concetto di diffusione. Come si è detto nel Capitolo 1, la membrana delle cellule non è impermeabile, al contrario, permette il passaggio di molte molecole, le quali la attraversano direttamente, nel caso siano piccole molecole apolari, o tramite proteine di membrana, come nel caso dell'acqua. Modellare questo comportamento in modo corretto è indispensabile per qualunque simulatore cellulare.

La simulazione consiste in un insieme di 5 000 cellule, disposte dentro un cerchio, con al centro una singola cellula, contenente 10 000 molecole. Ogni nodo ha al suo interno una sola reazione, la quale, invia la molecola ad un vicino, se essa è presente nel nodo, con concentrazione minima pari a uno. Il comportamento atteso è quello di una diffusione, fino ad ottenere una concentrazione costante della molecola in tutte le cellule. La simulazione, in modalità

grafica, è visibile in Figura 6.2. Il grafico sul numero di molecole è riportato in Figura 6.3.

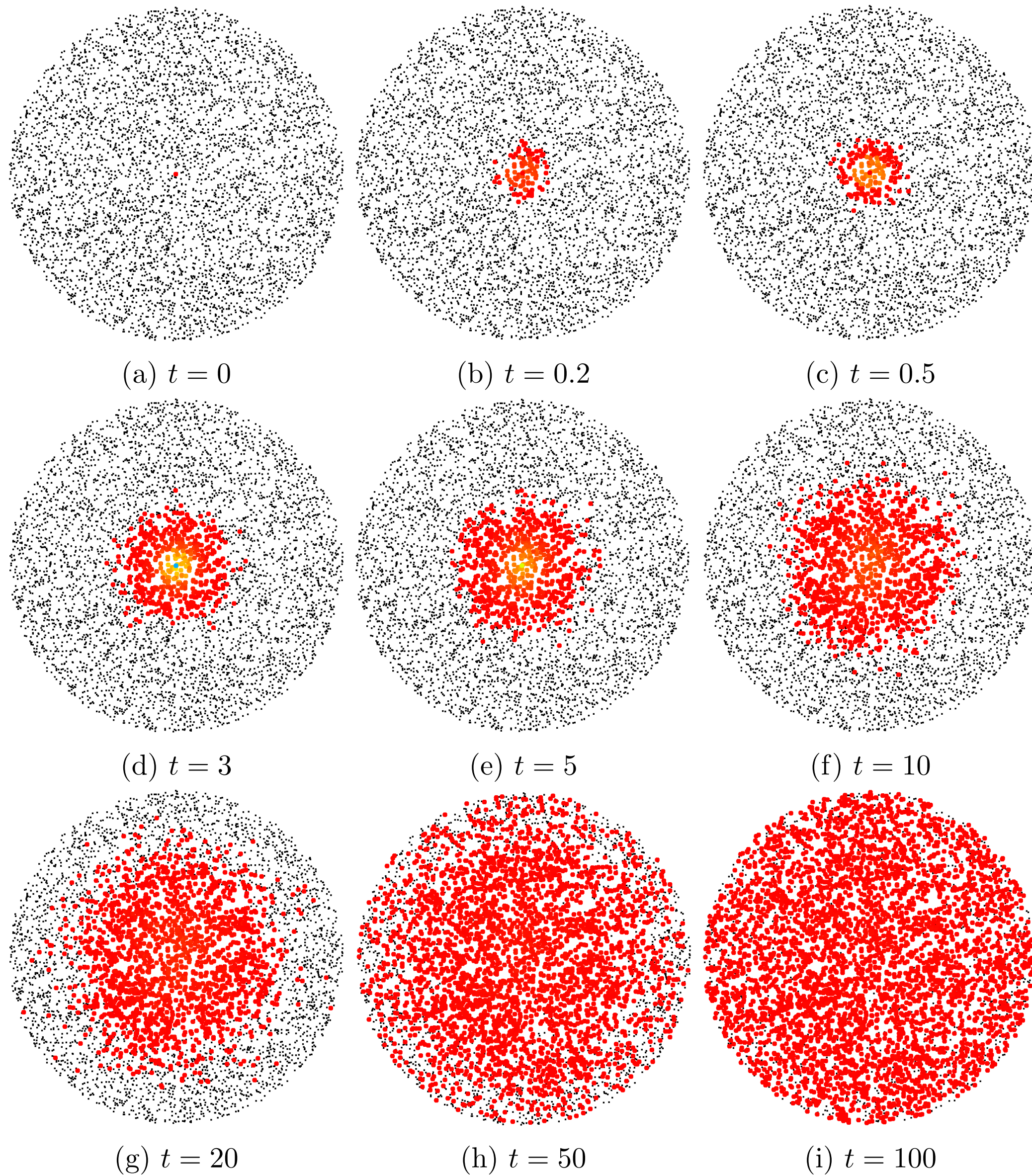


Figura 6.2: Diffusione di una molecola nel tempo, simulando il suo passaggio per la membrana delle cellule. Al tempo $t = 0$ solo una cellula contiene le molecole. Esse si diffondono sempre più, fino a presentarsi con concentrazione costante in tutti i nodi. Il colore giallo indica una maggiore concentrazione.

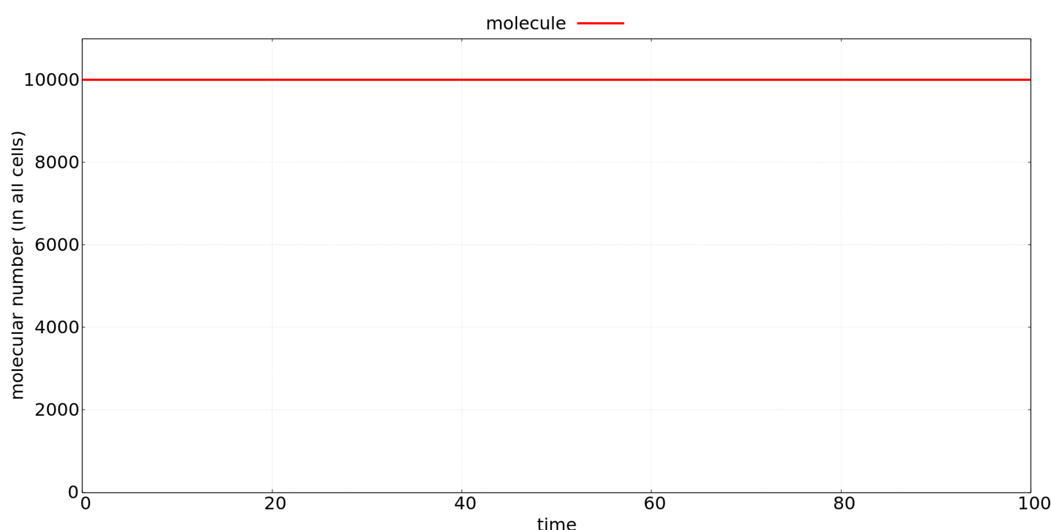


Figura 6.3: Grafico che mostra il numero totale di molecole presenti nel sistema, aggregando i dati di cinque simulazioni. Notare come il numero rimanga costante durante tutta la simulazione. Le molecole non vengono quindi perse o create durante il loro spostamento nelle cellule vicine.

6.3 Test sulle giunzioni

Il primo test sulle giunzioni fatto è stato relativo al corretto reinserimento delle molecole nelle rispettive cellule di appartenenza, una volta spezzata la giunzione. Esso è stato compiuto in un ambiente multi-cellulare, contenente qualche migliaio di cellule, ognuna con al suo interno le molecole `A` e `B`, usate per formare la giunzione. Si è monitorato l'andamento della concentrazione delle molecole all'interno di ogni cellula, riscontrando, per ogni simulazione, valori in linea con quando atteso. Le molecole vengono quindi reimmesse nelle cellule corrette, senza essere perse o eliminate.

Un altro test compiuto è stato quello sul movimento con giunzioni. Come si sa, se due cellule, muovendosi, rompono la relazione di vicinato, tutte le giunzioni in comune devono essere distrutte, e le molecole reinserite nelle cellule di origine. Il sistema descritto precedentemente è stato ampliato, aggiungendo anche il movimento. Tutte le simulazioni hanno dato esito positivo. I dati non sono riportati per non appesantire la trattazione, con grafici che si compongono solamente di linee costanti. Un aspetto più interessante riguarda il movimento come azione dell'utente. In Alchemist è infatti possibile selezionare una cellula, e spostarla nell'ambiente a proprio piacimento. Tutto quello detto per il movimento deve valere anche in questo caso, con la rottura delle giunzioni se non è più verificato il vicinato. Anche in questo caso i test compiuti sono stati

soddisfacenti, rompendo la giunzione e reinserendo le biomolecole nelle cellule di origine, anche a fronte di spostamenti manuali.

Un altro test compiuto riguarda la formazione delle giunzioni, in ambienti con alta o bassa concentrazione cellulare. Si è creato un insieme di 1000 cellule, disposte in un cerchio. Esse contengono ognuna 10 molecole `A` e 10 molecole `B`. Le giunzioni formate sono del tipo `A-B`. Quando una giunzione di questo tipo è presente, viene attivata la produzione di una molecola `S`, detta segnale, la quale permette di rilevare la presenza della giunzione. Più la concentrazione di questa molecola è elevata, maggiore saranno le giunzioni presenti. Intuitivamente, più le cellule sono vicine, più giunzioni devono formarsi tra loro. Per simulare questo comportamento si sono utilizzate due *linking rule* diverse. Nella prima due cellule sono considerate adiacenti solo se la loro distanza è minore o uguale a 0.1. Nella seconda esse sono vicine se distano al più 1. Ci si attende, quindi, una maggiore formazione di giunzioni nel secondo caso. Sono state compiute 10 simulazioni per ognuno dei due casi, ed i risultati aggregati. I grafici delle concentrazioni sono visibili in Figura 6.4.

Ovviamente questi test fatti sulle giunzioni, non sono simulazioni di sistemi reali, ma solo modellazione di concetti intuitivi, che devono essere sempre verificati. Questo perché la dinamica delle giunzioni è un aspetto molto complicato del comportamento cellulare. La loro formazione, e sopratutto le reazioni intracellulari a seguito di essa, sono complesse e numerosissime, cambiando in base al tipo di giunzione specifico. La modellazione di questi aspetti, prettamente biologici, non rientra negli obiettivi di questo elaborato di tesi, e quindi lasciato a sviluppi futuri.

6.4 Test di modelli reali

Simulare dinamiche cellulare reali non è un compito facile, sopratutto per l'elevatissimo numero di reazioni coinvolte. Il più delle volte, infatti, la simulazione si riduce alla modellazione di un singolo comportamento.

Come test su sistemi non banali, si è scelto il *repressilator* [12]. Esso è una rete di tre geni, che producono tre proteine, le quali reprimono i geni successivi. Uno schema è visibile in Figura 6.5. Il repressilator è una pietra miliare della *system biology*, poiché è stato il primo caso in cui una funzionalità è stata prima modellata e simulata "da zero", e poi impiantata in cellule vere e proprie, verificandone l'effettivo funzionamento. Esso consente la generazione di una oscillazione stabile, nella concentrazione delle tre proteine, all'interno della cellula, consentendo di ottenere risultati simili agli oscillatori elettronici.

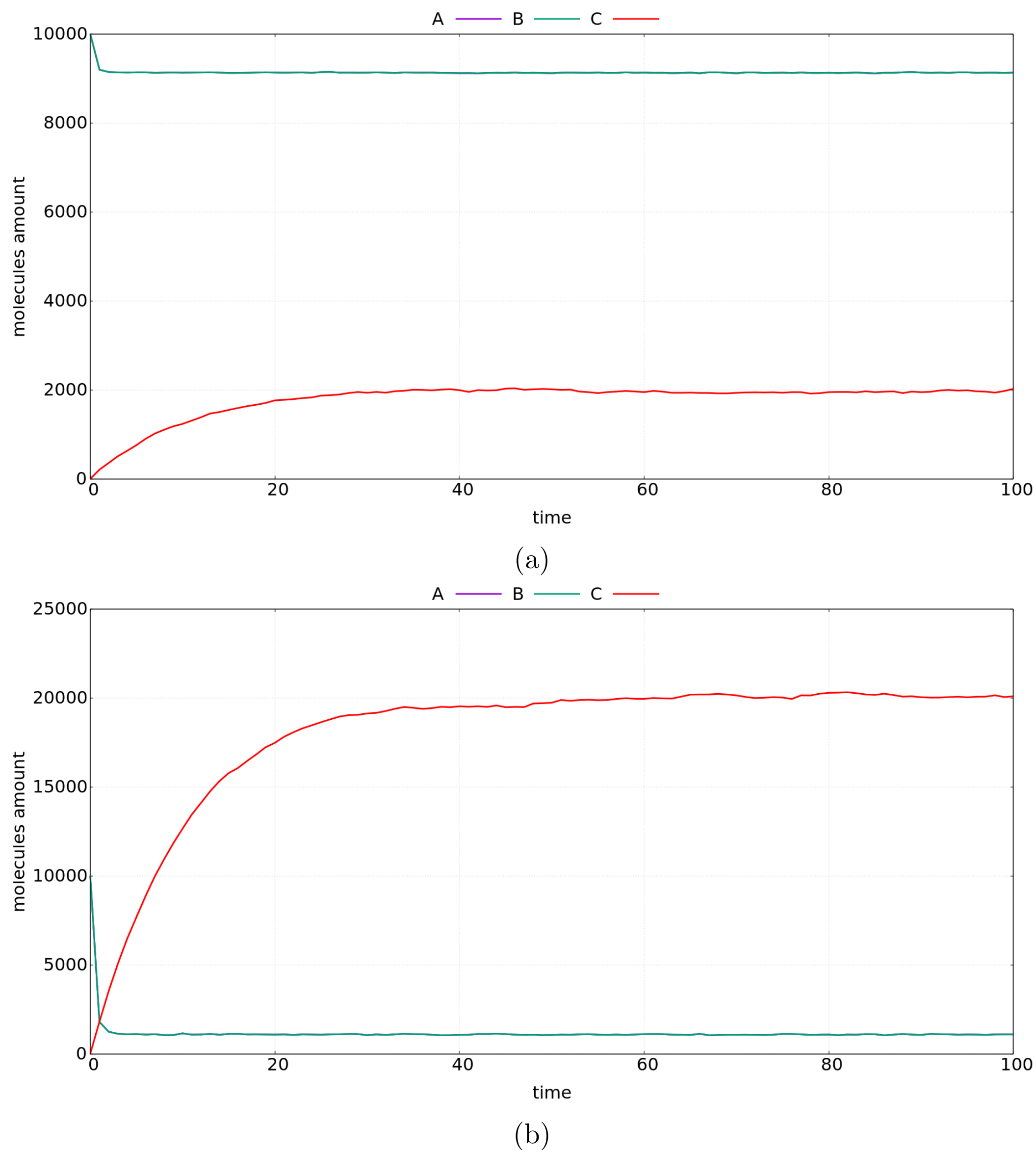


Figura 6.4: Simulazione della dinamica delle giunzioni. In (a) è presente una concentrazione cellulare minore, e le giunzioni sono formate in modo sporadico, come si può vedere dalla bassa presenza di molecole C. In (b), al contrario, la concentrazione di cellule è maggiore, e le giunzioni si formano in modo massiccio, come evidenziato dalla grande quantità di molecole C di segnalazione. Le molecole A e B sono usate per la formazione della giunzione, e il loro numero, ovviamente, rimane molto più alto in (a) rispetto a (b).

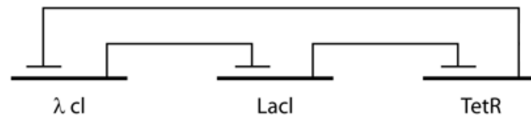


Figura 6.5: Il repressilator. Il gene λcI produce una proteina, la quale reprime il gene LacI. A sua volta, egli produce una proteina che reprime il gene TerR. Esso, tramite la produzione di una specifica proteina, reprime il primo gene: λcI. Questa rete consente un movimento oscillatorio della concentrazione delle tre proteine nella cellula. (fonte Wikipedia)

I dati inerenti alle reazioni chimiche ed i rate sono stati estrapolati da [13]. Esse sono quattro per ogni gene, per un totale di dodici reazioni. Uno schema è riportato di seguito:

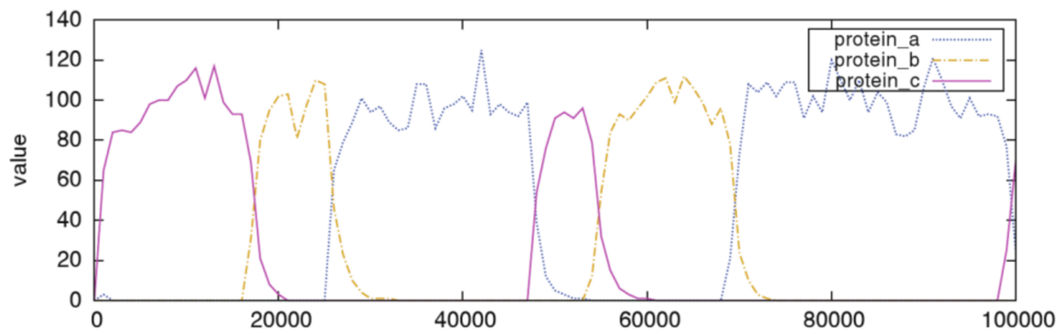
- $G_i \xrightarrow{r_1} G_i + P_i$ Generazione della proteina i, a partire dal corrispondente gene.
- $G_i + P_{i-1} \xrightarrow{r_2} G_i(block) + P_{i-1}$ Repressione del gene i, se è presente la proteina generata da gene precedente.
- $G_i(block) \xrightarrow{r_3} G_i$ Attivazione del gene i.
- $P_i \xrightarrow{r_4}$ Degradazione della proteina i.

Dove i rate sono: $r_1 = 0.1$, $r_2 = 1$, $R_3 = 10^{-4}$, $r_4 = 10^{-3}$. Le condizioni iniziali sono gene A attivo, mentre tutti gli altri bloccati.

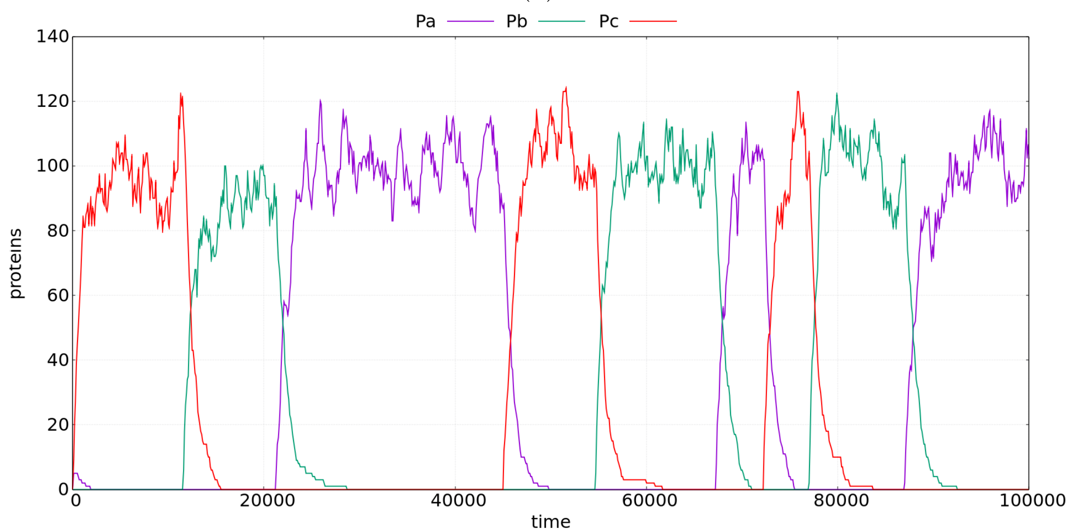
Il rate più veloce è inerente al blocco del gene, da parte della proteina prodotta dal gene precedente. In questo modo si riescono a riprodurre dei picchi di concentrazione delle proteine piuttosto alti, che calano subito non appena il gene corrispondente viene bloccato. Sono state fatte diverse simulazioni, con seed per i numeri casuali sempre diversi. Un esempio di una simulazione, confrontato con i dati estrapolati da [13] è visibile in Figura 6.6.

L'attivazione dei geni segue una distribuzione di probabilità casuale, quindi, in simulazioni fatte con diversi seed, possono verificarsi comportamenti leggermente diversi. Ad esempio un gene, durante una simulazione, può attivarsi prima che nella successiva, producendo periodi di stabilità più corti. A scopo di esempio, in figura Figura 6.7 è riportata la simulazione dello stesso sistema con seed pari a 0.

Proprio per questi motivi l'aggregazione dei dati risulta difficoltosa. In un istante di tempo, infatti, in alcune simulazioni potrebbe essere alta la produzione di proteine A, mentre in altre quelle di B o C, dovute a ritardi o anticipi nell'attivazione dei geni. Un grafico prodotto facendo la media di 30 simulazioni, con altrettanti seed, è visibile in Figura 6.8.



(a)



(b)

Figura 6.6: Raffronto tra i grafici ottenuti per la simulazione del repressilator, tramite un simulatore stocastico. In (a) sono visibili i dati sperimentali ottenuti in [13], mentre in (b) i dati ottenuti durante la simulazione con seed pari a 1. L'attivazione dei geni segue una distribuzione casuale, quindi ottenere due grafici completamente identici è quasi impossibile. Tuttavia essi sono molto simili, mostrando la corretta modellazione di questo comportamento cellulare.

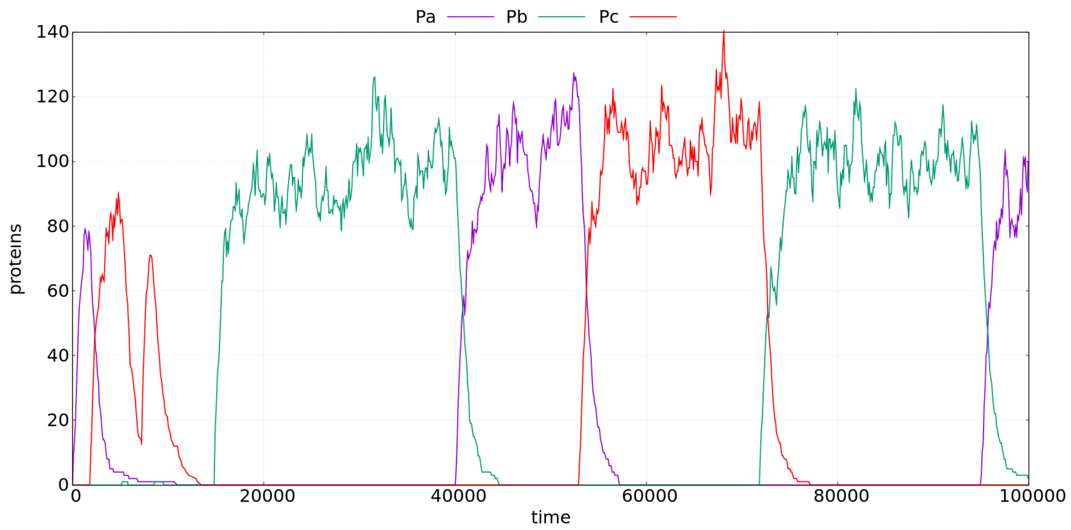


Figura 6.7: Simulazione del repressilator con seed pari a 0. Notare la differenza rispetto al grafico riportato sopra. Ciò è dovuto alle fluttuazione di probabilità, date dalla stocasticità del sistema. Il comportamento generale risulta però corretto.

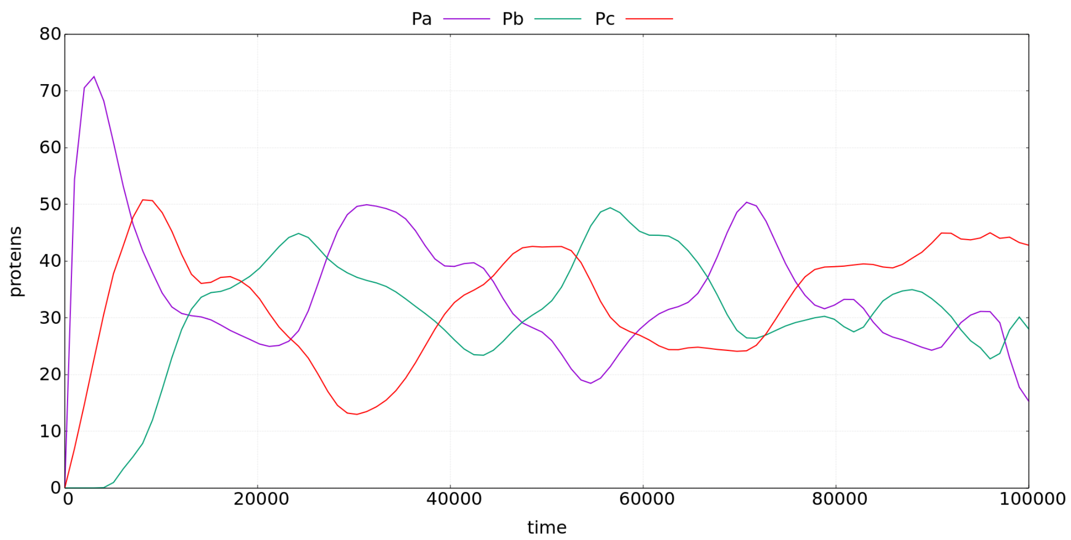


Figura 6.8: Aggregazione dei dati provenienti da 30 simulazioni distinte, dello stesso fenomeno. Notare come per le fluttuazioni dovute ai numeri casuali, usati per l'attivazione dei geni, non ci sia un comportamento di attivazione o repressione completa. Tuttavia sono ben visibili i picchi di produzione delle proteine, validando i risultati, anche a fronte di molte simulazioni differenti.

Repressilator spaziale

Negli ultimi anni, nel campo della *syntetic biology*, ha preso sempre più piede il concetto di coordinamento di intere colonie di cellule, attraverso l'azione di uno o più geni, prodotto da una piccola parte di esse, e poi diffusi a tutte le altre. Tramite tecniche di biologia molecolare, è possibile studiare "a tavolino" i comportamenti voluti, e poi iniettare geni specifici nelle cellule, per riprodurre i comportamenti progettati in colonie di cellule vive. Ad esempio si è riusciti a coordinare una colonia di batteri, affinché producesse onde di luce sincronizzate [14].

La nostra idea, è di sfruttare il repressilator, già testato precedentemente, a aggiungere la dimensione spaziale. Soltanto una cellula, chiamata *coordinator*, ha, al suo interno, la possibilità di attivare i geni. Ogni qualvolta che ciò accade, essa invia ai vicini una molecola di segnalazione, la quale segnala l'avvenuta attivazione del gene corrispondente. Una volta ricevuta la molecola, le cellule attivano il gene, e, a loro volta, inviano ai vicini molecole di segnalazione per quel gene. Si forma così un'onda di segnalazione, che, partendo dal coordinator, si espande a tutte le altre cellule.

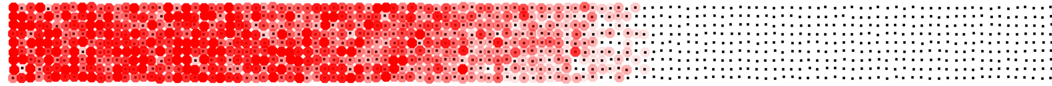
Si sono disposte le cellule secondo una griglia, lunga e stretta, in modo da favorire la generazione di onde di segnalazione. La cellula coordinator è stata posizionata all'estremo sinistro della griglia. Tutte le cellule, a parte quella di coordinazione, hanno tutti i geni bloccati, all'inizio della simulazione, mentre il coordinator ha attivo solo il gene A. In totale le cellule sono circa 1300. Alcune immagini della simulazione sono presentate in Figura 6.9.

Si è quindi riusciti a produrre un sistema di coordinazione tra cellule, basato su una sola cellula *master*, in modo simile a quanto avviene con le reti elettroniche. Notare come questo modello non sia solamente un mero esercizio teorico, poiché la coordinazione programmata di gruppi di cellule sta prendendo sempre più piede nel campo della biologia molecolare [14].

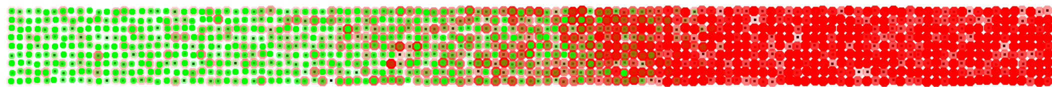
Il grafico dell'andamento della concentrazione delle proteine in tutte le cellule, è visibile in Figura 6.10.

6.5 Valutazione finale

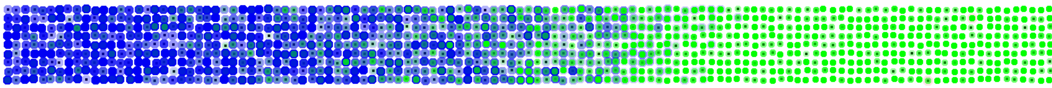
Tutti i test, compiuti sull'incarnazione Biochemistry, hanno dato esito positivo, validando di fatto la correttezza del progetto. I dati ottenuti sono risultati ottimi, confrontabili perfettamente con quelli presenti in letteratura, usati come modello. I test sono stati compiuti in modo sincrono con lo sviluppo, permettendo di intercettate preventivamente errori o comportamenti indesiderati. Questo ha fatto in modo di procedere con uno sviluppo del soft-



(a)



(b)



(c)

Figura 6.9: Simulazione del repressilator spaziale, descritto precedentemente. La lunghezza della griglia è stata lievemente ridotta per garantire immagini migliori. L'intensità dei colori indica la concentrazione delle proteine. Inizialmente i nodi non contengono alcuna proteina. Il nodo coordinator, avendo in gene A attivo, dalle condizioni iniziali, inizia a produrre la proteina A (rosso), e segnala ai vicini, tramite una proteina di segnalazione, l'attivazione del gene A. Essi, attivandolo, cominciano a produrre la proteina A, e, a loro volta, segnalano ai vicini. Si forma quindi un'onda di segnalazione (a). Quando nella cellula coordinator viene attivato il gene C, esso produce proteine C (verde), le quali reprimono il gene A, attivo in precedenza. Esso viene segnalato alle cellule adiacenti, che attivano il gene C e segnalano ai vicini. Le proteine A prodotte in precedenza, decano per effetto della loro degradazione. Si forma quindi un'altra onda di segnalazione, che comunica l'attivazione del gene C (b). Alla stessa maniera si procede per il gene B (blu, Fig. (c)), il quale fa partire la produzione di proteine B, che reprimono il gene C. Si riparte poi da A, in un ciclo infinito di cambi di stato.

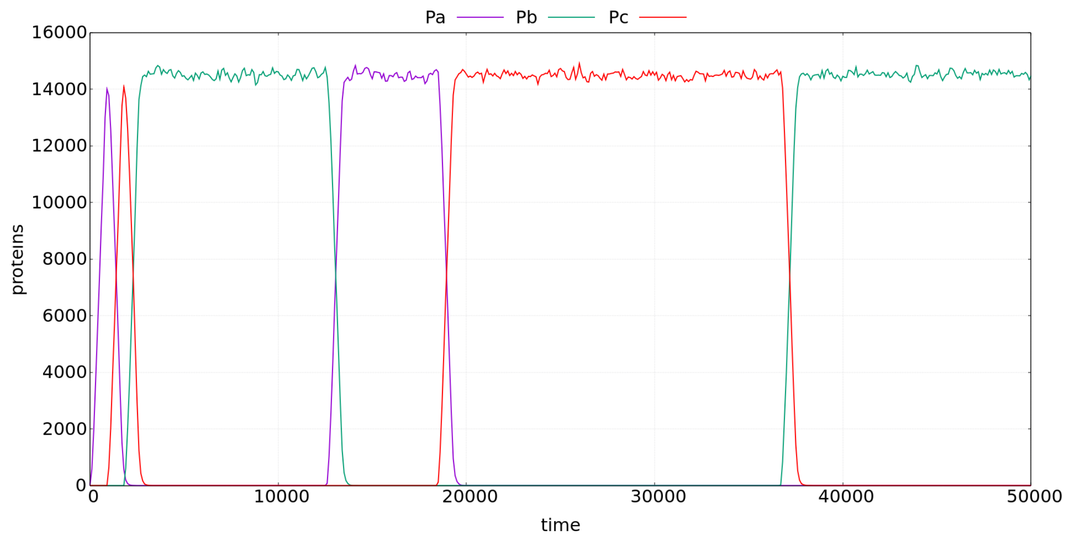


Figura 6.10: Grafico che mostra l'andamento della concentrazione delle proteine, sommate per tutte le cellule. Esso è un repressilator quasi perfetto, con pochissime fluttuazioni. Notare la diversa durata dei cicli, dovuta all'attivazione dei geni in modo casuale.

ware più veloce, procedendo agli step successivi solo se tutto risultata corretto e realisticamente attendibile.

La modellazione di sistemi molto complessi, come, ad esempio, alcune dinamiche cellulari, non sono parte di questo lavoro di tesi, poiché più vicine alla chimica e alla biologia. Tuttavia, la scrittura del modello per le simulazioni fatte è risultata semplice, ponendo le basi per l'uso del simulatore anche a persone con scarse conoscenze biologiche.

Anche se i test non rispecchiano dinamiche cellulari complesse, si è riusciti comunque a modellare comportamenti all'avanguardia nel campo della *syntetic biology*, come la coordinazione di un insieme di cellule. In particolare per il test sul repressilator spaziale, non sono state trovate fonti sull'esecuzione di simulazioni stocastiche su sistemi simili. Si è quindi dimostrato come l'applicazione sviluppata, non solo consenta la simulazione di sistemi noti, ma anche quella di sistemi creati *ad-hoc*, per la simulazione di particolari dinamiche, anche non esistenti direttamente in natura.

Conclusioni

Giunti al termine del lungo percorso, che dalla definizione dei requisiti ha portato all'implementazione dell'incarnazione Biochemistry sul simulatore Alchemist, mi posso dire pienamente soddisfatto del lavoro svolto, e dell'applicazione sviluppata. Partendo dai requisiti proposti è stato compiuto un processo a più fasi, guidato dalle regola alla base dell'ingegneria dei sistemi software. Si è proceduto all'analisi dei requisiti, e allo studio, approfondito, dei sistemi utilizzati. Questo è stato particolarmente stimolante, sopratutto nella parte inerente agli algoritmi usati per la simulazione di sistemi naturali. Particolare attenzione è stata posta in fase di design: sia per costruire un sistema facilmente estendibile e modificabile, sia per intercettare, nella maniera corretta, tutti i concetti alla base del simulatore Alchemist, ed implementarli, poi, al meglio.

Tutti i requisiti dati sono stati soddisfatti, producendo un'applicazione che permette la simulazione di dinamiche cellulari e multi-cellulari. I risultati dei test compiuti sono stati, infatti, al di sopra delle aspettative, producendo dati praticamente identici a quelli presenti in letteratura. L'incarnazione permette anche la simulazione di sistemi non realmente esistenti, ma progettati e simulati *in silico*. Quest'ultima parte è sicuramente il risultato di maggior rilievo ottenuto in questo elaborato di tesi. Come dimostrato dal sempre maggior numero di ricerche nel campo della *syntetic biology*, la simulazione di dinamiche cellulari, che poi saranno portate su cellule vive, è uno dei campi di maggior fermento nel più vasto campo della bio-ingegneria.

Ovviamente non si è implementato un simulatore multi-cellulare completo, ovvero che fornisce la possibilità di modellare qualsiasi comportamento. Fenomeni come la diffusione di molecole nell'ambiente, o la genesi e distruzione di cellule, non sono state intercettate in questo elaborato, ma saranno oggetto di studi ed estensioni future. Anche i test dovranno essere estesi, producendo dinamiche sempre più complesse e realistiche. Ciò è consentito dalla facile estensibilità del sistema, punto cardine che ha seguito tutte le fasi, dall'analisi all'implementazione. In particolare, nel breve periodo, saranno modellate, ed implementate, la diffusione ambientale di molecole, la quale consente di simulare in modo più realistico ambienti multi-cellulari, ed una progettazione più

realistica del movimento cellulare.

Una delle dinamiche che sarà oggetto di modellazione nell'immediato futuro è l'implementazione di un test, atto a simulare il *genetic clock* presente in [14]. Sono già stati svolti lavori preliminari, ma non sono stati inclusi in questa relazione poiché incompleti. La maggiore difficoltà nel riprodurre questo comportamento sta nel passaggio tra modello deterministico, sotto forma di equazioni differenziali, e modello stocastico, usato dal simulatore. Non sono stati trovati riferimenti in letteratura riguardo il modello stocastico di [14], il che costringe ad un lavoro di *fitting* dei parametri, non sempre immediato. La maggiore difficoltà è dovuta al concetto di ritardo (*delay*) tra la creazione di una molecola, e la possibilità che essa sia usata da un'altra reazione. Ad esempio una proteina, una volta creata, deve passare attraverso varie fasi, come il *folding*, prima di poter essere utilizzata. Questo concetto è assente in Alchemist, in cui una molecola non appena è creata è utilizzabile immediatamente.

Come detto, questo progetto pone una prima, fondamentale, pietra nella costruzione del simulatore biochimico Biochemistry. Molte altre persone parteciperanno al progetto, migliorandolo ed espandendolo. L'obiettivo finale è quello di simulare fenomeni biologici estremamente complessi, come la morfogenesi e lo sviluppo embrionale.

Il lungo cammino, che ha portato all'implementazione di questo progetto di tesi, è stato molto interessante e stimolante. Come mai prima d'ora la mia conoscenza è stata ampliata, con i più disparati argomenti, i quali spaziano da tecniche allo stato dell'arte nell'ingegneria del software, fino a concetti propri della biologia molecolare. Tutti gli argomenti trattati sono sempre stati tra i miei maggiori interessi, anche se non avevo mai avuto l'opportunità di approfondirli. Esco da questi mesi di duro lavoro sicuramente arricchito, sia sul piano culturale, sia nella consapevolezza dei miei mezzi e dei miei punti di forza.

Bibliografia

[1] S. E. Woosley et al. “Type Ia supernovae: Advances in large scale simulation”. In: *Journal of Physics: Conference Series* 180.1 (2009).

[2] C. F. McDonagh et al. “Antitumor activity of a novel bispecific antibody that targets the ErbB2/ErbB3 oncogenic unit and inhibits heregulin-induced activation of ErbB3”. In: *Molecular Cancer Therapeutics* 11.3 (2012), pp. 582–593.

[3] Jonathan R. Karr et al. “A Whole-Cell Computational Model Predicts Phenotype from Genotype”. In: *Cell* 150.2 (2012), pp. 389–401.

[4] Patrick Forterre. “Defining Life: The Virus Viewpoint”. In: *Origins of Life and Evolution of Biospheres* 40.2 (2010), pp. 151–160.

[5] *Simulatore Alchemist*. URL: `https://alchemistsimulator.github.io`.

[6] Daniel T. Gillespie. “Exact stochastic simulation of coupled chemical reactions.” In: *The Journal of Physical Chemistry* 81.25 (1977), pp. 2340–2361.

[7] M. A. Gibson e Bruck J. “Effcient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels.” In: *The Journal of Physical Chemistry A* 104.9 (2000), pp. 1876–1889.

[8] Yang Cao, Hong Li e Linda Petzold. “Efficient formulation of the stochastic simulation algorithm for chemically reacting systems”. In: *The Journal of Chemical Physics* 121.9 (2004), pp. 4059–4067.

[9] Alexander Slepoy, Aidan P. Thompson e Steven J. Plimpton. “A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks”. In: *The Journal of Chemical Physics* 128.20 (2008).

[10] *Rate equation*. URL: `https://en.wikipedia.org/wiki/Rate_equation`.

[11] Daniel T. Gillespie. “Simulation Methods in Systems Biology”. In: *Formal Methods for Computational Systems Biology*. A cura di Marco Bernardo, Pierpaolo Degano e Gianluigi Zavattaro. 2008, pp. 125–167.

[12] Michael B. Elowitz e Stanislas Leibler. "A synthetic oscillatory network of transcriptional regulators". In: *Nature* 403.6767 (2000), pp. 335–338.

[13] F. Liu e M. Heiner. "Petri Nets for Modelling and Analyzing Biochemical Reaction Networks". In: *Approaches in Integrative Bioinformatics*. A cura di Ming Chen e Ralf Hofestädt. 2014, pp. 245–272.

[14] Tal Danino et al. "A synchronized quorum of genetic clocks". In: *Nature* 463.7279 (2010), pp. 326–330.