ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA -
SCIENZA E INGEGNERIA

Corso di Laurea in Ingegneria e Scienze Informatiche

RI-PROGETTAZIONE DEL MODULO DI ESPORTAZIONE DATI DEL SIMULATORE ALCHEMIST

Tesi di Laurea in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. Danilo Pianini

Presentata da
Acampora Andrea

Correlatore
Prof. Mirko Viroli

Anno Accademico 2020/2021

Sommario

Le simulazioni computerizzate sono uno strumento vitale utilizzato nell'analisi del comportamento di sistemi complessi. Ad oggi le simulazioni sono diventate parte integrante dell'ingegneria dei sistemi computazionali, principalmente durante la fase di progettazione per testare, convalidare e prevedere il comportamento del sistema prima di iniziare l'effettiva fase di dispiegamento. In generale, le simulazioni computerizzate richiedono la definizione di un modello del dominio, la sua implementazione eseguibile, l'effettiva esecuzione e la produzione dei dati di output. L'esportazione dei dati è un aspetto chiave nel contesto delle simulazioni computerizzate, in quanto rappresenta l'output dell'operazione di simulazione. Lo scopo di questo lavoro è quello di analizzare il dominio della generazione di dati tramite simulazione, analizzare lo stato dell'arte e infine applicare quanto appreso nella ri-progettazione di un modulo di esportazione dati per il simulatore Alchemist. Fra gli altri, il nuovo modulo dovrà superare le limitazioni del sistema originale in termini di flessibilità e prestazioni.

Indice

Capitolo 1

Contesto e Motivazioni

1.1 Le simulazioni computerizzate

Le simulazioni computerizzate sono uno strumento vitale utilizzato nell'analisi del comportamento di sistemi complessi. Ad oggi le simulazioni sono diventate parte integrante dell'ingegneria dei sistemi computazionali, principalmente durante la fase di progettazione per testare, convalidare e prevedere il comportamento del sistema prima di iniziare l'effettiva fase di dispiegamento. In generale, possiamo intendere le simulazioni computerizzate [21] come un metodo per studiare sistemi. Questo metodo, o processo [7] si componde delle seguenti fasi:

1. la presenza di un *fenomeno* che si intende osservare;
2. l'implementazione di un modello matematico rappresentativo del fenomeno;
3. la trasformazione in un modello eseguibile;
4. l'indagine del fenomeno di partenza attraverso il modello opportunamente creato. In questa fase, durante l'esecuzione della simulazione, verrà aggiornato lo stato del sistema e delle variabili presenti nel modello seguendo un opportuno criterio per lo scorrere del tempo;
5. l'analisi delle informazioni ottenute e generate durante la simulazione.

Le simulazioni computerizzate, inoltre, fungono da strumento pedagogico [9]. Anche quando un problema presenta una soluzione analitica, possiamo utilizzare le simulazioni per comprendere ed analizzare il processo che l'ha prodotta. Nel corso degli anni le simulazioni computerizzate sono state responsabili di una serie di progressi e sono diventate indispensabili nella modellazione di molti sistemi sia naturali (fisica [15], chimica [22], biologia [19]) che economici [23] e sociali [20]. Inoltre, le simulazioni sono ampiamente utilizzate in tutti i rami dell'ingegneria

al fine di ottenere una maggiore comprensione e conoscenza dei sistemi appena menzionati. In sintesi, le principali motivazioni [24] che portano all'utilizzo delle simulazioni computerizzate sono:

- In un sistema complesso non sempre è possibile trovare una descrizione analitica;
- Un esperimento in un contesto reale e non digitalizzato è spesso meno efficiente, più costoso, e solitamente più complesso;
- In molti contesti le simulazioni permettono di analizzare e comprendere un sistema reale a cui è impossibile accedere;
- Esistono scenari in cui vengono simulate le condizioni che portano alla distruzione di un sistema. Nella maggioranza dei casi, la distruzione di un sistema reale non è permessa e di conseguenza le simulazioni computerizzate rimangono l'unica soluzione;
- Alcuni esperimenti, per svariate ragioni, necessitano di poter fermare l'esecuzione. Questo aspetto è certamente possibile in una simulazione computerizzata, mentre può risultare complicato in contesti reali.

Le simulazioni necessitano di un set di dati di partenza che vengono forniti attraverso sensori e attuatori, file di configurazione, valori generati da un qualche processo oppure valori di output di altre simulazioni.

1.2 Generazione di dati tramite simulazioni

Molte simulazioni computerizzate, a prescindere dal contesto in cui operano, generano continuamente grandi quantità di dati. Gli utilizzi e le funzionalità [8] dei dati generati dalle simulazioni possono essere raggruppati come segue:

1. Fornire un resoconto della simulazione effettuata e di ciò che la computazione ha generato;
2. Valutare e confrontare le prestazioni dei modelli adottati;
3. Nelle simulazione stocastiche, in cui sono presenti parametri stocastici, è necessario eseguire delle analisi per verificare che i valori generati siano statisticamente significativi.

Sebbene essi siano estremamente rilevanti attualmente non è presente uno standard per l'esportazione. Le ragioni risiedono nel fatto che i dati da esportare variano in base al dominio in cui opera la simulazione, ed essendo esse utilizzate in numerosi campi risulta impossibile adottare un formato standard che sia adeguato a tutti i contesti. A fronte di ciò, spesso si ricorre all'utilizzo di formati personalizzati, i quali sono creati ad-hoc sul dominio dei dati che si intende esportare.

1.2.1 Esempi in letteratura

Il simulatore **Repast** [17], nato all'interno dell'Università di Chicago, è uno strumento open-source per la modellazione e simulazione basata su agenti. Per quanto riguarda l'esportazione dei dati il simulatore Repast permette all'utente di generare un file di testo con i dati di output e supporta inoltre la visualizzazione di essi tramite grafici. **Neuron** [14] è un simulatore utilizzato nell'ambito della neuroscienza per studiare le attività elettriche delle reti neuronali. Neuron esporta i dati in diversi file in formato *High Oriented Calculator*(HOC). In particolare, genererà un file contente i soli parametri della simulazione, un file contenente i dati della simulazione ed un file da utilizzare per visualizzare grafici in 3D. **NetLogo** [2], un ambiente completo per la programmazione multi-agente, non permette un esportazione diretta dei dati della simulazione ma consente di visualizzarli tramite grafici e, solo al termine dell'esecuzione, è possibile esportare i dati dei grafici su un file in formato CSV. Il simulatore ad eventi **NS-3** [13], utilizzato per scopi di ricerca nell'ambito delle reti internet, prevede un modulo di import/export che si occupa di tradurre i dati della simulazione nel formato *Functional Mock-Up Interface*(FMI) [5]. **Omnet++** [16], una piattaforma per simulazioni ad eventi di reti internet, utilizza il formato *Network Description*(NED) per la configurazione delle simulazioni. In aggiunta, per esportare i dati mette a disposizione un modulo che consente di convertire dal formato *NED* al formato *JSON*, in modo da agevolare la comprensione dei dati di output.

Capitolo 2

Il simulatore Alchemist

Alchemist è un simulatore stocastico *open-source* nato all'interno dell'Università di Bologna, che permette la simulazione di scenari inerenti la computazione pervasiva, aggregata ed ispirata alla natura.

2.1 Analisi del modello di dominio

Il dominio del simulatore Alchemist, rappresentato in fig. 2.1, è composto dalle seguenti entità:

- **Molecola**: può essere associata ad un concetto di variabile in un linguaggio di programmazione imperativo. Ogni *molecola* è descritta non solo da una *concentrazione*, ma da un insieme di proprietà in modo che lo stato del sistema possa essere complesso e strutturato.
- **Concentrazione**: è il valore di una determinata *molecola*. Può essere associato al valore di una variabile in un linguaggio di programmazione.
- **Nodo**: contenitore di *reazioni* e *molecole* situato all'interno di un *ambiente*.
- **Ambiente**: rappresenta un'astrazione dello spazio. E' un contenitore di *nodi* e ha la funzione di:
  1. comunicare la posizione dei nodi nello spazio.
  2. comunicare la distanza tra due nodi.
  3. funge da supporto per i nodi in movimento.
- **Regola di associazione**: una funzione di un determinato *ambiente* che associa ogni *nodo* ad un *vicinato*.

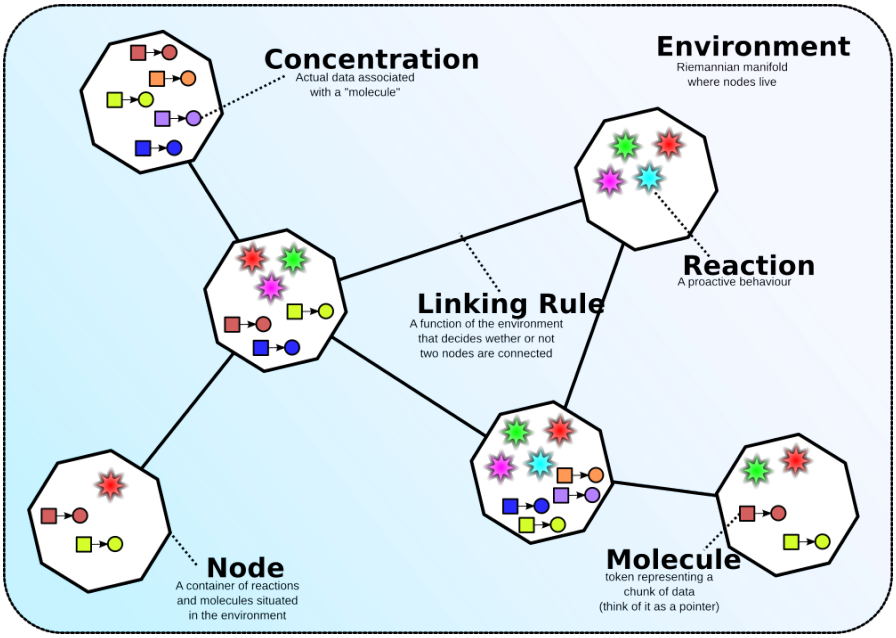

Figura 2.1: Il modello computazionale di Alchemist: uno spazio continuo contenente nodi. Ciascun nodo è programmato con un set di reazioni e contiene un set di molecole.

- **Vicinato**: un'entità composta da un nodo e un set di nodi vicini.
- **Reazione**: Il concetto di *reazione* è da considerarsi molto più elaborato di quello utilizzato in chimica: in questo caso, si può considerare come un insieme di *condizioni* sullo stato del sistema, che qualora dovessero risultare vere innescherebbero l'esecuzione di un insieme di *azioni*. La frequenza di accadimento di una reazione può dipendere dal valore di ciascuna *condizione*, da un tasso statico, da una distribuzione temporale e da un'equazione che a partire dal tasso statico e dai valori delle condizioni restituisce un tasso istantaneo. (Vedere fig. 2.2 per comprendere il funzionamento). Ogni *nodo* contiene un'insieme anche vuoto di reazioni.
- **Condizione**: funzione booleana che prende in ingresso un ambiente e determina se una *reazione* deve essere eseguita.
- **Azione**: modella un cambiamento nell'ambiente.

Le implementazioni del meta-modello sono definite come incarnazioni ed hanno l'obbligo di ridefinire il concetto di concentrazione delle molecole. Al momento le incarnazioni presenti sono: *Protelis*, *Sapere*, *Biochemistry*, *Scafi*.

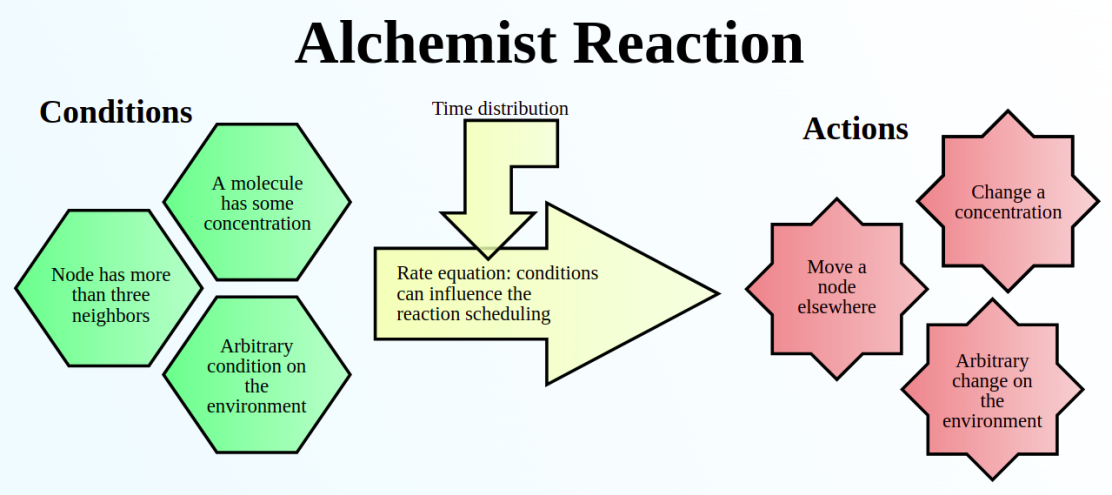

Figura 2.2: Rappresentazione grafica del funzionamento di una reazione.

2.2 Funzionalità ed utilizzo del simulatore

Le simulazioni in Alchemist vengono configurate all'interno di file YAML [4], un formato per la serializzazione di dati facilmente leggibile. Nei file di configurazione è necessario specificare quali classi e parametri utilizzare. Le entità vengono identificate tramite chiavi, al livello zero troviamo ad esempio:

- **incarnation**: l'unica chiave obbligatoria. Serve a specificare quale incarnazione si vuole utilizzare;
- **seed**: specifica i semi da utilizzare per la generazione di numeri casuali;
- **variables**: una lista di variabili aggiuntive;
- **environment**: la tipologia di ambiente dentro al quale svolgere la simulazione;
- **network-model**: permette di scegliere come devono essere collegati tra loro i nodi a seconda della classe;
- **export**: permette di configurare e scegliere i dati della simulazioneda esportare;
- **deployment**: permette di definire quali sono i nodi del sistema e dove si trovano.

Per una descrizione più dettagliata delle chiavi si faccia riferimento al sito di Alchemist[1]. Il simulatore può funzionare in due modalità: interattiva o batch. Nella prima viene eseguita una sola simulazione per volta mentre in modalità batch possono essere eseguite molteplici simulazioni a seconda dei valori delle variabili impostati dall'utente. Le variabili permettono quindi di impostare dei set di valori al posto di un valore singolo. In caso di più variabili con più valori, il simulatore eseguirà una simulazione per ogni possibile combinazione delle stesse, ossia eseguirà un numero di simulazioni pari alla cardinalità del prodotto cartesiano delle cardinalità dei valori delle stesse. L'elevato grado di libertà nella configurazione delle simulazioni permette ad Alchemist una serie di funzionalità, tra le quali:

- l'esecuzione di programmi Scafi[2].
- l'esecuzione di programmi Protelis[3].
- l'esecuzione di programmi SAPERE[4].
- la simulazione di ambienti bidimensionali.
- la simulazione di pedoni con modelli cognitivi.
- l'esecuzione di simulazioni in ambienti interni importando immagini in bianco e nero e ambienti esterni come mappe fornendo anche supporto alla navigazione.
- la simulazione di reti di smart-cameras.
- l'esecuzione di simulazioni in ambienti distribuiti.

2.3 Riproducibilità degli esperimenti scientifici

Un fenomeno che non può, almeno in linea di principio, essere riprodotto o osservato a piacimento non può essere materia di indagine scientifica. Nonostante Alchemist utilizzi un modello computazionale puramente stocastico, uno dei punti chiave del simulatore è quello di garantire la riproducibilità degli esperimenti scientifici. Nel simulatore è infatti essenziale anche a fini di debug riuscire a garantire delle simulazioni sempre uguali tra loro partendo da una stessa configurazione nel file YAML.

[1] `https://alchemistsimulator.github.io/wiki/usage/yaml/`

[2] `https://scafi.github.io/`

[3] `https://protelis.github.io/`

[4] `https://www.unibo.it/en/research/projects-and-initiatives/Unibo-Projects-under-7th-Framework-Programme/cooperation-1/information-and-communication-technology-ict-1/sapere`

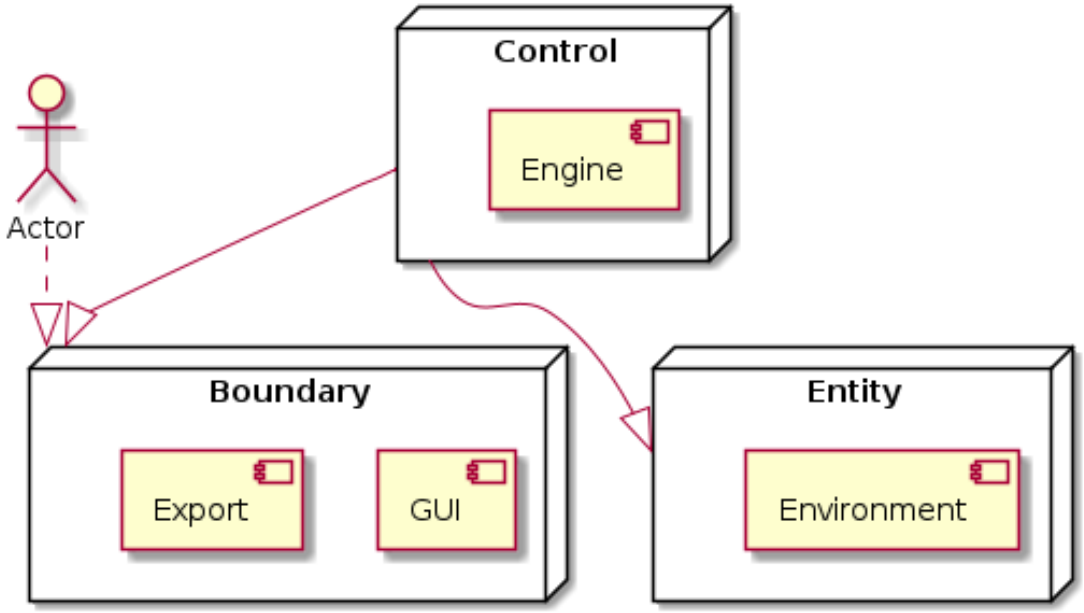

Figura 2.3: Diagramma dell'architettura ECB di Alchemist.

2.4 Analisi dell'architettura di esportazione dati preesistente

Alchemist utilizza un'architettura di tipo ECB [12] (*Entity Control Boundary).* Il pattern architetturale ECB può essere considerato un'estensione del pattern MVC [18] dove le *entity* corrispondono al *model* ed i *boundary* alla *view*:

- **Entity**: entità passiva, rappresentativa di una qualche parte del domonio;
- **Control**: funge da intermediario tra *entity* e *boundary*;
- **Boundary**: incapsula le interazioni con attori esterni.

Nel pattern la responsabilità di ogni classe è associata al ruolo che assume nel diagramma dei casi d'uso, nello specifico il *boundary* trasmette l'input dell'utente al controller e ne gestisce l'output mentre il controller contiene l'implementazione del motore del simulatore attraverso il quale modifica lo stato delle entità del dominio. Il *controller* comunica lo stato del sistema ai *boundary* tramite l'interfaccia `OutputMonitor` per esportarlo ad una GUI oppure su file. La fig. 2.3 mostra come viene applicatto il pattern nell'architettura del simulatore Alchemist.

L'attuale sistema di Export è basato sulla scrittura dei dati della simulazione su un file di testo in formato CSV. La classe responsabile dell'esportazione dei dati

è l'`Exporter` che implementa direttamente il boundary `OutputMonitor`. Nella fase di setup viene aggiunto all'`Engine`, ossia al *Controller*, il boundary rappresentato dalla classe `Exporter`. In questo modo il controller notificherà all'`Exporter` per ogni step della simulazione i relativi dati riguardanti le reazioni avvenute e quest'ultimo provvederà ad elaborarli ed esportarli su file. Nell'elaborazione ed estrazione dei dati l'`Exporter` utilizza l'interfaccia `Extractor`, la quale prevede una serie di implementazioni a seconda della tipologia di dato da estrarre. Nella fig. 2.4 viene mostrato il diagramma delle classi dell'architettura preesistente del simulatore Alchemist.

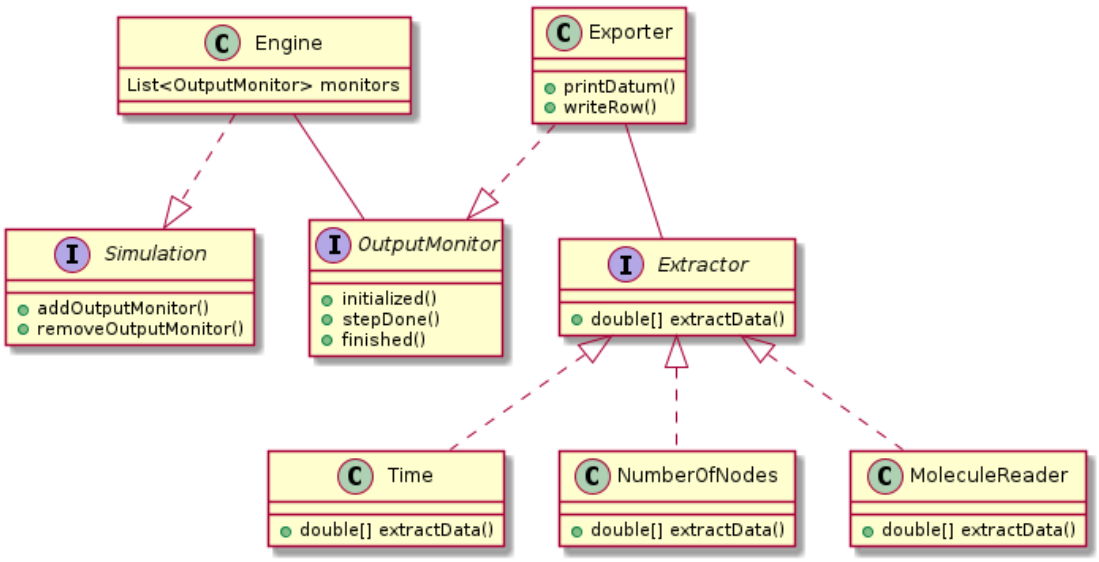

Figura 2.4: schema UML della architettura di export di Alchemist esistente.

La scelta dei dati che si intende esportare avviene all'interno del file di configurazione YAML con la chiave *export*. Inoltre è possibile aggiungere dei filtri e aggregatori sui dati come minimo, massimo, media, varianza come nel seguente esempio:

```yaml
export:
    - time
    - molecule: "default_module:default_program"
    aggregators: [mean, max, min, variance, median]
    value-filter: onlyfinite
```

La linea di comando presenta i seguenti parametri:

- `-e, --export`: parametro obbligatorio se si intende effettuare la fase di export. Contiene una stringa formata dal percorso in cui verrà generato il file di output.
- `-i, --interval`: parametro opzionale, rappresenta l'intervallo di campionamento ossia con quale frequenza esportare i dati della simulazione.

Capitolo 3

Un nuovo sistema di esportazione dati orientato alla flessibilità

In questo capitolo verrà analizzato il contributo fornito al progetto, elencando i requisiti necessari e analizzando il processo di realizzazione degli stessi. L'obiettivo del progetto consiste nell'efficientamento dell'attuale sistema di esportazione dati del simulatore Alchemist implementando un modulo in grado di esportare i dati con diverse tecnologie in modo dinamico a seconda della configurazione scelta dall'utente.

3.1 Analisi dei requisiti

In questa sezione si elencano i requisiti funzionali e non funzionali che verranno realizzati.

3.1.1 Requisiti funzionali

A livello funzionale, la nuova architettura dovrà soddisfare i seguenti requisiti:

- **aumento della flessibilità**: una delle caratteristiche fondamentali del nuovo modulo di esportazione dati è appunto la flessibilità. L'utente potrà scegliere una tra le varie tecnologie disponibili per esportare dati oppure potrà scegliere di utilizzarne molteplici contemporaneamente. In relazioni ai dati da esportare, il nuovo sistema dovrà lasciare all'utente la libertà di scegliere quali dati assegnare ad ogni tipologia di esportatore.
- **aggiunta di target per l'esportazione**: partendo da un sistema con un unico target di esportazione dati, ossia un file in formato CSV, il nuovo mo-

dulo aggiungerà un'altra destinazione per i dati di output della simulazione. Un esempio può essere sicuramente l'utilizzo di un database esterno.

- **aggiunta di esportatori**: a fronte di ogni nuovo target per i dati generati dalla simulazione dovrà essere presente il rispettivo esportatore. In questo modo l'utente potrà scegliere in completa libertà quale esportatore utilizzare e di conseguenza con quale tecnologia esportare i dati.

- **compatibilità con l'architettura preesistente**: il nuovo sistema di esportazione dati dovrà essere perfettamente compatibile con l'architettura del simulatore, ossia dovrà aggiungere delle nuove funzionalità senza alterare in alcun modo quelle già presenti.

- **configurazione degli esportatori interamente nel file YAML**: la scelta di quale tecnologia utilizzare, ossia se esportare su file oppure utilizzare un database esterno, verrà effettuata nella fase di configurazione della simulazione all'interno del file *YAML*.

A fronte del requisito di aggiungere dei nuovi target viene scelto di utilizzare come destinazione per i dati di output un database esterno. Le motivazioni che hanno portato all'utilizzo di un database sono le seguenti:

- **Accesso efficiente ai dati**: utilizzando un database sarà possibile eseguire delle interrogazioni per accedere a dei dati specifici senza dover necessariamente leggere l'intero file CSV.

- **Persistenza e affidabilità**: una volta memorizzati all'interno del database sarà possibile accedere ai dati in qualsiasi momento.

- **Condivisione**: un database può essere condiviso nel senso che applicazioni diverse possono accedere, secondo opportune modalità, a dati comuni. Questo risulta molto utile nel contesto del simulatore nel caso in cui si voglia lanciare una simulazione su server differenti per bilanciare il carico di lavoro.

- **Scalabilità**: in casi in cui l'esportazione prevede grandi quantità di dati vengono generati file di grosse dimensioni e difficili da leggere mentre i database sono ottimizzati [6] per gestire i dati in memoria.

3.1.2 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà non comportamentali che il sistema deve possedere, come efficienza, affidabilità, sicurezza, performance, ma anche caratteristiche del processo di sviluppo e caratteristiche esterne. Per quanto riguarda la performance, sarà sicuramente necessario rispettare o eventualmente

migliorare il tempo di esecuzione della fase di esportazione dati. Inoltre, il nuovo target dovrà apportare dei miglioramenti nella gestione dei dati di output ossia dovrà ridurre l'occupazione totale di essi.

3.2 Analisi e ricerca delle nuove tecnologie

A fronte del nuovo utilizzo di un database esterno in fase di analisi è stata svolta una ricerca per scegliere la tecnologia che si adattasse maggiormente alle specifiche richieste. Sono emersi due database *NoSQL* : **Redis** e **MongoDB**.

3.2.1 Redis

Redis [1] è un datastore in memoria rapido, *open source* e di tipo chiave-valore. Tutti i dati risiedono *in-memory*, a differenza dei database che memorizzano i dati su disco o SSD. Eliminando l'esigenza di accedere ai dischi, i datastore in-memory come *Redis* evitano i ritardi dovuti ai tempi di ricerca e sono in grado di accedere ai dati in pochi microsecondi. *Redis* presenta strutture dati versatili, elevata disponibilità, geospazialità, scripting Lua, transazioni, persistenza su disco e supporto a cluster. Tutte le operazioni sono atomiche, quindi in caso di accessi concorrenti da parte di più client, i dati forniti risulteranno sempre consistenti. *Redis* ha conseguito un risultato epocale con la release della versione 5.0 che comprende una serie di funzioni avanzate e miglioramenti. La vera notizia, in questo caso, è l'introduzione di *Streams*, la prima struttura dati completamente nuova in *Redis* dai tempi di HyperLogLog. Questa release ha aggiunto anche altri comandi per *Sorted Sets* e nuove funzionalità per le API dei moduli. Riassumendo le funzioni che offre *Redis* sono le seguenti:

- **Datastore in memoria**
  Tutti i dati di Redis si trovano nella memoria principale del server, a differenza di quanto avviene nei database come PostgreSQL, Cassandra, MongoDB e altri, che memorizzano la maggior parte dei dati su disco o su SSD. Nei database tradizionali basati su disco, i dati vengono trasferiti da e verso lo storage per ogni operazione, mentre i datastore in memoria come Redis non prevedono questa operazione. Pertanto, questo consente loro di supportare una quantità di operazioni di ordine di grandezza superiore e tempi di risposta più rapidi. Il risultato è costituito da prestazioni incredibilmente elevate con tempi medi di lettura e scrittura inferiori al millisecondo e supporto per milioni di operazioni al secondo.

- **Strutture dati flessibili**
  A differenza di datastore chiave-valore che offrono strutture dati limitate,

Redis offre strutture dati adatte a diverse applicazioni. I tipi di dati Redis includono ad esempio stringhe,liste sets, sorted sets, hash, bitmap.

- **Portabilità e compatibilità**
  Sono disponibili più di cento client open source. Tra i linguaggi supportati sono presenti Java, Python, PHP, C, C++, JavaScript, Node.js, Ruby, R, Go e molti altri.

- **Replica e persistenza**
  Redis impiega un'architettura primary-replica [3] e supporta la replica asincrona, in cui i dati vengono replicati su diversi server appositi. In questo modo è possibile ottenere migliori prestazioni in lettura (poiché le richieste vengono suddivise tra i diversi server) e ripristino in caso di interruzione del server principale. Per la persistenza, Redis supporta backup point-in-time (copia su disco del dataset Redis).

3.2.2 MongoDB

MongoDB [10] è uno dei più noti database non relazionali (NoSQL). Si tratta di una soluzione orientata ai documenti, che sfrutta il formato JSON per la memorizzazione e la rappresentazione dei dati. Il documento è fondamentalmente un albero che può contenere molti dati, anche annidati. MongoDB si adatta a molti contesti, in generale quando si manipolano grandi quantità di dati eterogenei e senza uno schema. Non è invece opportuno quando si devono gestire molte relazioni tra oggetti, e si vuole garantire l'integrità referenziale tra essi. Il modello di un documento di MongoDB è facile da comprendere, inoltre sono disponibili driver per più di dieci linguaggi, oltre alle varie decine sviluppati dalla community. Tra le caratteristiche offerte da MongoDB possiamo trovare:

- **Document-Oriented**
  il protagonista del database è il Document (il dato) e non la struttura e le relazioni che ci sono fra le varie tabelle.

- **Flessibilità**
  l'essere "Schema-less" permette di poter svincolare la struttura della tabella (Collection) dall'effettiva istanza del Document; se successivamente alla creazione iniziale si necessita di utilizzare campi in più e/o di diverso tipo, non c'è bisogno di aggiornare tutta la struttura della Collection e dei Documents già presenti.

- **Scalabilità orizzontale**
  MongoDB (e in generale i vari database NoSQL) utilizzano il meccanismo

di Sharding che distribuisce i dati su più server in modo automatico senza che l'applicazione sappia la composizione del cluster; se abbiamo bisogno di migliorare le performance, basta aggiungere delle macchine in più (MongoDB "sfrutta" la RAM dei server).

- **Velocità**
  MongoDB è molto performante [11] nella gestione dei Documents, anche in presenza di grandi quantità di dati.

A partire dalla realease 3.5 dei driver di MongoDB per Java è stato introdotto un oggetto chiamato *CodecRegistry* che permette di convertire oggetti Java direttamente in documenti JSON. In questo modo si evita di creare manualmente il documento ed è possibile aggiungere oggetti al database in modo più facilmente.

3.2.3 Comparazione e scelta

Una volta analizzate entrambe le tecnologie si è scelto di implementare un sistema basato sul database **MongoDB**, pur senza escludere eventuali successive implementazioni con *Redis*. Le ragioni di questa scelta sono:

- Una caratteristica principale di Redis è il fatto di salvare i dati in *RAM*, fornendo prestazioni elevate. Il miglioramento delle prestazioni nel salvataggio dei dati non è un requisito rilevante per la nostra architettura.

- Il modello dei dati di MongoDB è simile al modello del dominio del simulatore. Ciascuna collezione può essere tradotta in una simulazione, mentre un documento chè formato da una mappa rappresenta l'insieme dei dati che vogliamo esportare ed i relativi valori.

- MongoDB mette a disposizione a livello gratuito dei cluster in cloud su cui sviluppare dei database. Questa aspetto risulta interessante nel contesto del simulatore ad esempio nella situazione in cui vengono lanciate simulazioni da server differenti e non in locale.

3.3 Progettazione

Una volta effettuata l'analisi dei requisiti e scelta la tecnologia da adottare il passo successivo riguarda la progettazione della struttura del software.

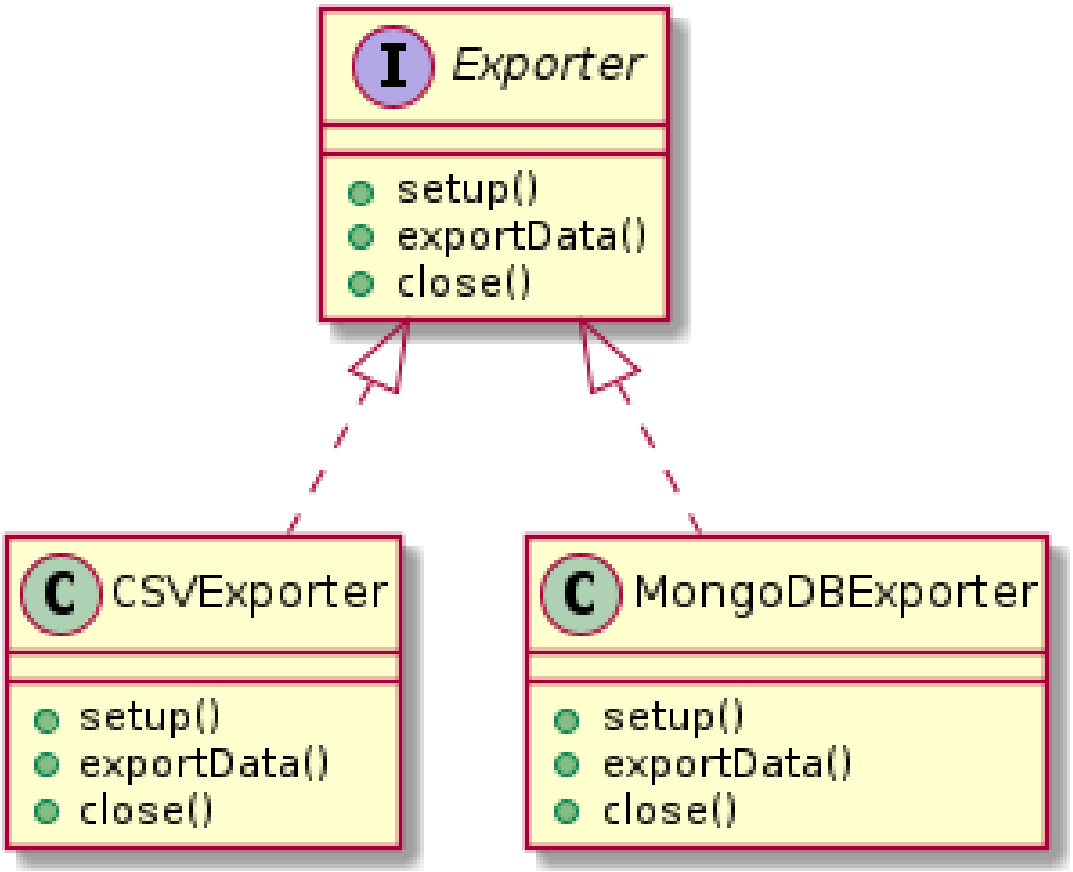

Figura 3.1: Schema della nuova interfaccia Exporter.

3.3.1 Design della nuova architettura

In un'architettura per l'esportazione di dati con molteplici esportatori risulta necessaria la presenza di un'interfaccia generica a cui fare riferimento. A fronte di ciò viene introdotta l'interfaccia `Exporter`, attorno la quale ruota l'intero sistema di export e che rappresenta un generico esportatore per il simulatore. L'interfaccia conterrà dei metodi per esportare dati e dei metodi per inizializzare i relativi ambienti. Saranno presenti diverse implementazioni dell'interfaccia e ciascuna di esse rappresenterà un alternativo metodo di export. Nello specifico vengono realizzate le seguenti implementazioni:

- `CSVExporter`: si tratta di un'estensione della classe originale `Exporter` e si occupa di esportare i dati su un file in formato *CSV* compatibile con il precedente sistema.

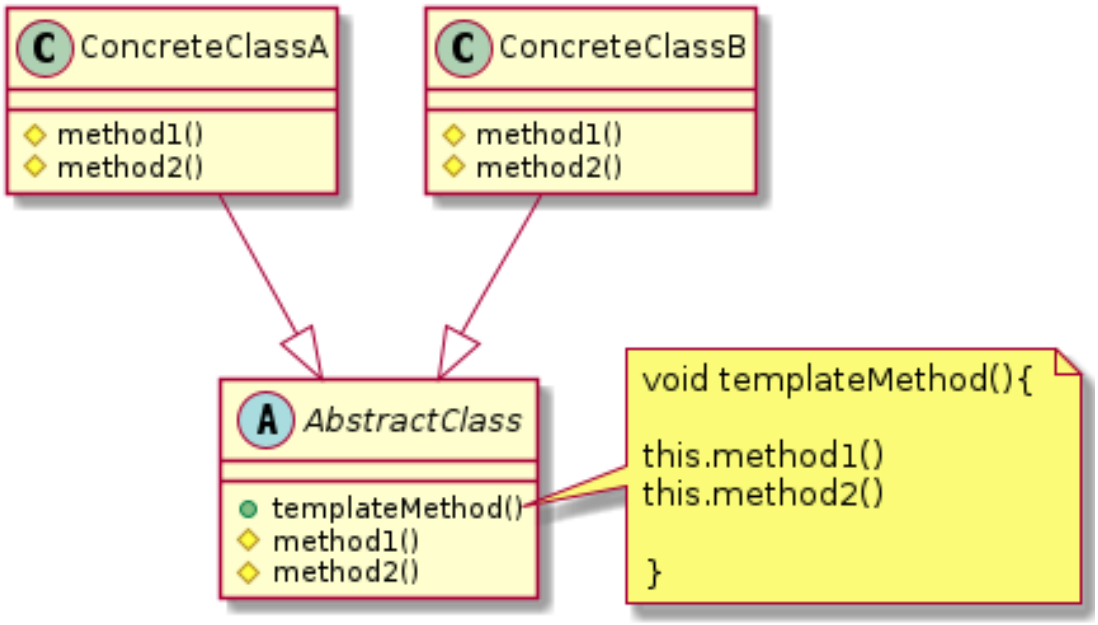

Figura 3.2: Diagramma delle classi del design pattern Template Method.

- `MongoDBExporter`: si occupa di esportare i dati della simulazione sul database NoSQl *MongoDB*. La classe dovrà essere in grado di esportare i dati indipendentemente dalla locazione del database: verranno eseguiti dei test sia su delle istanze del database in locale che su soluzioni in cloud.

Gli esportatori presenti nella nuova architettura svolgono un insieme di operazioni come ad esempio l'inizializzazione dell'ambiente di export che sono in realtà comuni ad entrambi. A fronte di ciò risulta estremamente comodo creare un'implementazione astratta di un esportatore che si occupa di svolgere solo ed esclusivamente le operazioni comuni in modo da evitare inutili ripetizioni di codice. Viene quindi creata la classe `AbstractExporter` la quale conterrà i metodi comuni a tutte le implementazioni dell'interfaccia e in particolare si occuperà di contenere gli oggetti necessari a entrambe per estrarre valori a partire dalle entità del simulatore. Nella realizzazione della classe si è reso utile l'utilizzo del design pattern **Template Method**. Il pattern comportamentale *Template Method* viene utilizzato per definire la struttura di un algoritmo delegando alcuni passi di dettaglio alle sottoclassi. Questo pattern nasce dall'esigenza di specificare l'ordine delle operazioni da effettuare ma di delegare alle sottoclassi l'implementazione di alcune operazioni. Pertanto il metodo che definisce l'algoritmo viene implementato nella superclasse mentre i metodi che definiscono i comportamenti di dettaglio vengono dichiarati astratti nella superclasse ed implementati nelle sottoclassi. Questo pattern è

composto dalle seguenti entità:

- **AbstractClass**: definisce il metodo concreto ed i metodi primitivi astratti. Il metodo concreto richiama i metodi primitivi implementati nelle sottoclassi.
- **ConcreteClass**: implementa i metodi primitivi per svolgere i passi specifici dell'algoritmo ed eventualmente i metodi hook.

L'utilizzo di questo pattern consente di:

- implementare una sola volta la parte "immutata" dell'algoritmo e di consentire alle sottoclassi di implementare il comportamento delle parti "variabili";
- individuare comportamenti comuni delle sottoclassi e "promuoverli" a comportamenti della superclasse in modo da evitare la duplicazione di codice: un esempio di refactoring di codice.

Attraverso il metodo template *update()* la classe astratta delegherà alle sue sottoclassi l'implementazione dello stesso a seconda della tipologia di esportazione che esse rappresentano.

A fronte del requisito funzionale di poter scegliere il tipo di esportazione direttamente nel file di configurazione, viene modifcata la sintassi della chiave `export`. Un esempio della nuova sintassi è il seguente:

```yaml
export:
    - type: AlchemistClassicCSV
      parameters:
        filename: "/my/data/folder"
        appendTime: true
      data:
        - time
        - molecule
        value-filter : onlyfinite

    - type: MongoDBExporter
      parameters:
        uri: "mongodb@127.0.0.1:8080"
        dbName: "alchemist-test"
        interval: 2.5
      data:
        - time
        - molecule
        aggregators: [mean, max, min, variance, median]
```

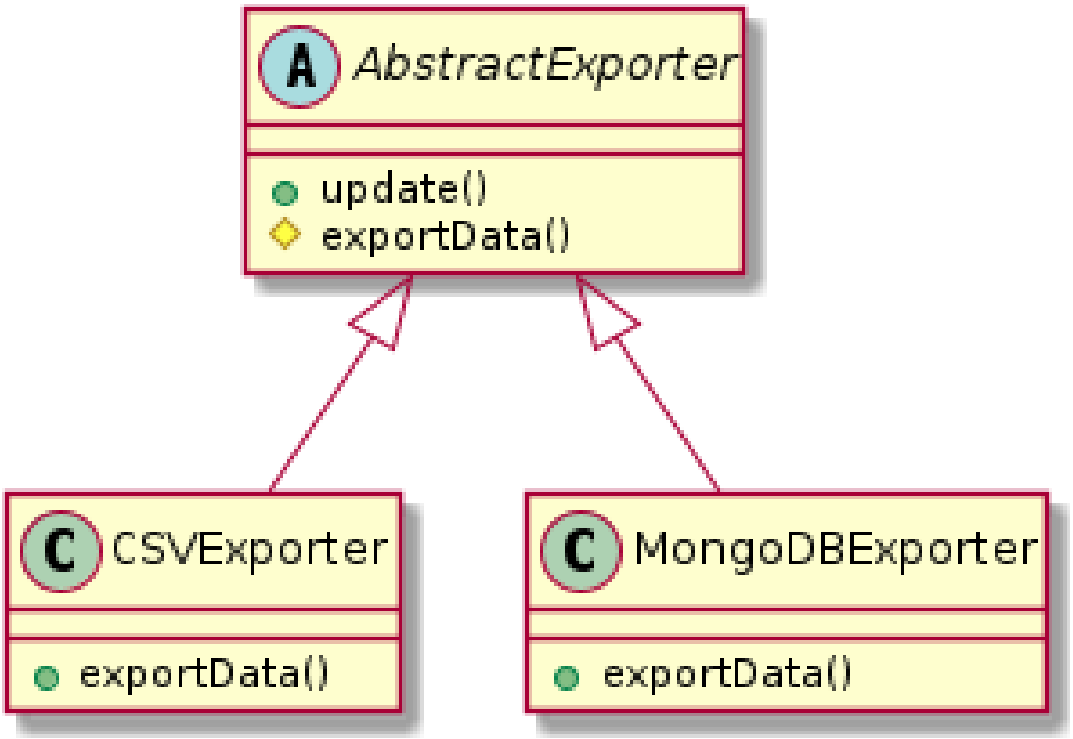

Figura 3.3: Schema del pattern Template Method della nuova architettura.

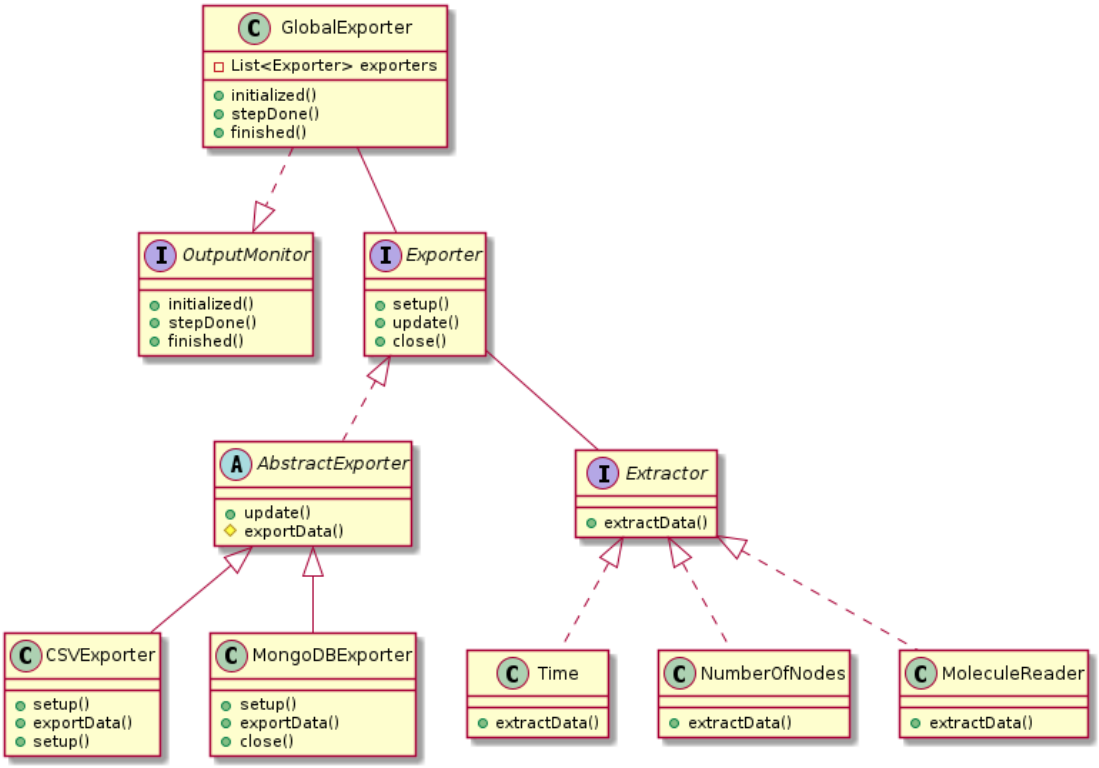

Figura 3.4: Diagramma delle classi della nuova architettura.

In particolare, è possibile dichiarare una lista di esportatori e, per ciascuno di essi, specificarne la tipologia. Tra i parametri aggiunti possiamo trovare i valori dei costruttori degli esportatori come ad esempio *fileNameRoot*, *uri* e *interval* e la chiave *data* che rappresenta la sezione nella quale si dichiarano i dati da esportare. Alcuni parametri saranno opzionali in quanto avranno dei valori di default, altri invece saranno obbligatori in quanto indispensabili per poter istanziare correttamente gli esportatori.
La nuova interfaccia `Exporter` non viene considerata in senso stretto un *boundary* dell'applicazione ossia non implementa direttamente l'interfaccia `OutputMonitor`. Le ragioni di questa scelta nascono del fatto che viene considerato come *boundary* l'intero sistema di export e non ogni singolo esportatore. A fronte di ciò viene creata la classe `GlobalExporter`. La classe implementa il boundary `OutputMonitor` e contiene al proprio interno la lista degli esportatori scelti nel file di configurazione. Durante l'esecuzione il motore del simulatore notificherà alla classe ad ogni passo lo stato della simulazione, quest'ultima provvederà a fare eseguire l'export ad ognuno dei suoi esportatori. La fig. 3.4 mostra una visione globale della nuova architettura.

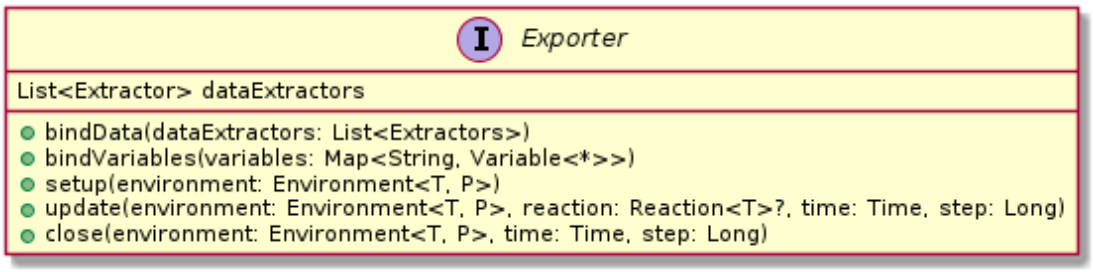

Figura 3.5: Interfaccia Exporter.

3.3.2 Design dettagliato

In questa sezione approfondiremo alcuni elementi di design con maggior dettaglio.

Exporter

In un generico esportatore per il simulatore risulta di primaria importanza contenere dei metodi per elaborare e convertire i dati forniti dalla simulazione in un formato esportabile. Tutto ciò viene svolto dall'interfaccia `Extractor`. Nello specifico l'interfaccia conterrà una serie di implementazioni a seconda della tipologia di dato da estrarre. Durante la preparazione della simulazione verrà assegnata all'esportatore una lista di estrattori tramite il metodo `bindData`. In aggiunta agli estrattori di dati verranno comunicate all'esportatore anche le variabili utilizzate tramite il metodo `bindVariables`. La fase di preparazione dell'ambiente di esportazione viene eseguita all'interno del metodo `setup` che verrà invocato prima dell'inizio della simulazione. Per terminare l'esportazione in modo corretto è presente il metodo `close` il quale verrà invocato al termine della simulazione. Ad ogni passo della simulazione verrà invocato il metodo `update` nel quale la classe astratta controllerà se far esportare i dati a seconda dell'intervallo di campionamento scelto.

CSVExporter

La classe `CSVExporter` ha il compito di esportare i dati generati dalla simulazione su un file in formato CSV. Per utilizzare questa tipologia di esportatore è possibile fornire alcune informazioni aggiuntive nel file di configurazione. In particolare è possibile indicare il nome del file che verrà generato grazie al parametro `fileNameRoot` e l'intervallo di campionamento tramite il parametro `interval`. Inoltre è possibile specificare una destinazione per la cartella contenente il file di output tramite il parametro `path`. Se non viene indicata nessuna destinazione, il simulatore esporterà il file all'interno di una cartella temporanea. Esistono scenari

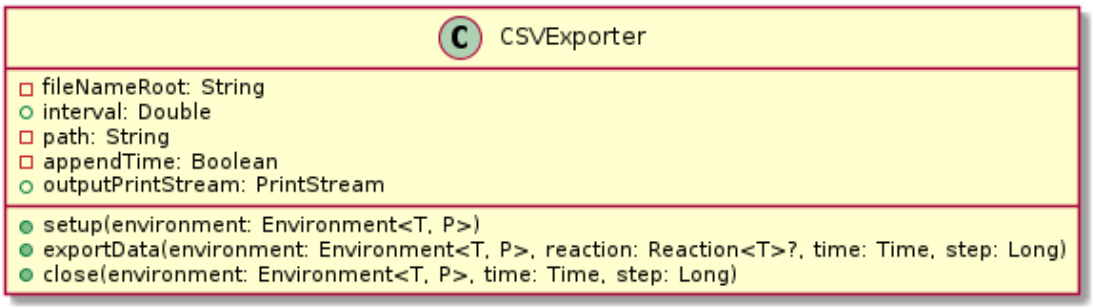

Figura 3.6: classe CSVExporter.

in cui si intende confrontare il file di output con quello della precedente esecuzione mentre in altre situazioni l'unico file di interesse è quello della simulazione più recente. Per supportare entrambi i casi viene utilizzato il parametro `appendTime`. In relazione ai parametri scelti, la stringa finale contenente la destinazione del file di output sarà salvata nella proprietà `exportDestination`. All'interno del metodo `setup()` la classe si occuperà di creare la cartella in cui generare il file se essa non è presente. Successivamente verrà istanziato un oggetto della classe `PrintStream` che avrà il compito di aprire uno stream in scrittura sul file di output. Le prime informazioni esportate riguardano i nomi delle variabili presenti nella simulazione e i nomi delle colonne dei dati che si andranno ad esportare. Il metodo `exportData()` conterrà l'effettiva implementazione dell'esportazione dei dati sul file. Per ciascun elemento appartenente alla lista interna di estrattori verranno esportati i dati generati in relazione a i valori correnti dello step della simulazione. Nel metodo `close()` l'exporter si occuperà di scrivere sul file un messaggio contenente l'orario di fine simulazione e di terminare in modo corretto l'esecuzione dello stream in scrittura.

MongoDBExporter

La classe `MongoDBExporter` si occupa di esportare i dati della simulazione sul database NoSQl *MongoDB*. Se si intende utilizzare questa tipologia di esportatore è obbligatorio indicare nel file di configurazione l'indirizzo in cui è presente un istanza di *Mongo* tramite il parametro `uri`. In aggiunta è possibile specificare il nome del database su cui esportare i dati tramite il parametro `dbName`. Se non viene indicato alcun valore verrà utilizzato il database di default chiamato *test*. In comune con il `CSVExporter` è possibile specificare l'intervallo di campionamento con il parametro `interval` e si può scegliere se sovrascrivere i dati nel database ad ogni esecuzione con il parametro `appendTime`. Tutto ciò che riguarda il dominio di *MongoDB* ossia i vari oggetti e collezioni per connettersi ed interagire

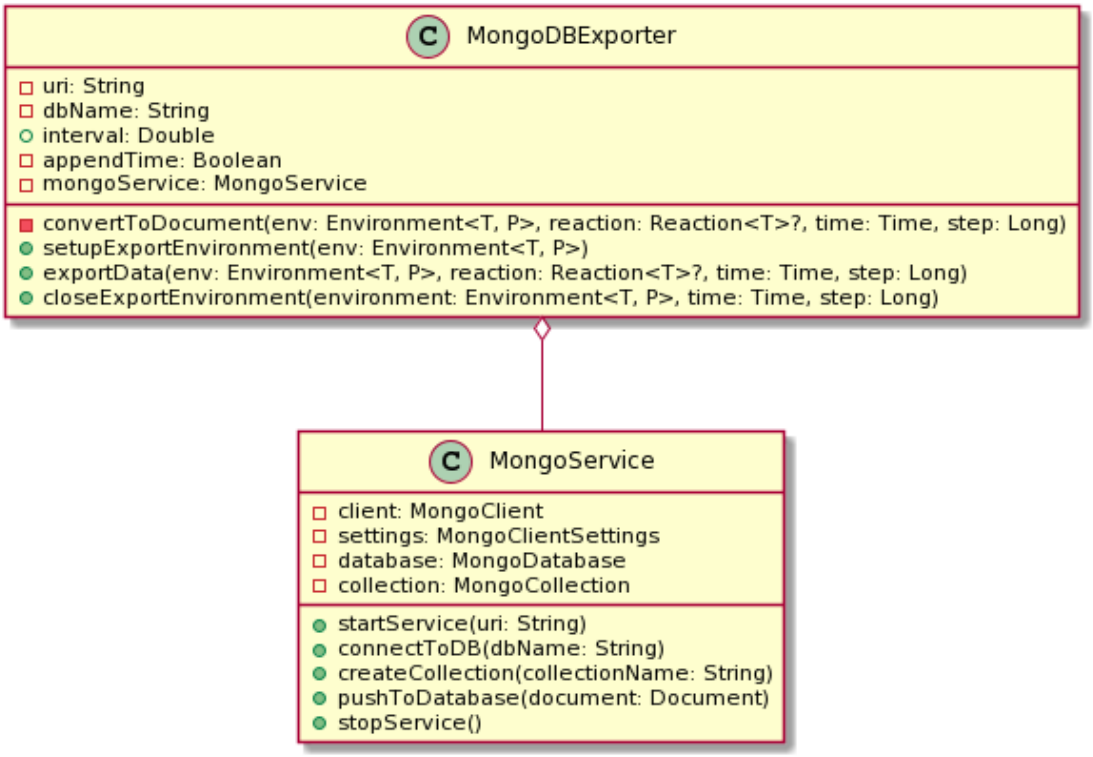

Figura 3.7: schema dell'interazione tra le classi MongoDBExporter e MongoService.

con il database saranno presenti unicamente all'interno della classe `MongoService` che verrà istanziata come campo dell'esportatore. Nel metodo `setup()`, a partire dalla stringa di connessione, avviene la connessione al database e la creazione di una collezione che rappresenta un'insieme di documenti per la simulazione corrente. Per fare ciò, l'esportatore utilizza i metodi `startService()`, `connectToDB()` e `createCollection()` della classe *MongoService*. Il metodo `exportData()` utilizza il metodo privato `convertToDocument()` per ottenere un documento in formato BSON[1] ed una volta ottenuto utilizza il metodo `pushToDatabase()` del *service* per aggiungere il documento alla collezione nel database. Il metodo privato `convertToDocument()` si occupa di generare un documento BSON a partire dalla lista interna di estrattori di dati. Infine nel metodo `close()` viene terminata in modo corretto l'esecuzione del *service*.

3.4 Implementazione

L'insieme delle classi e test sviluppati si trovano sotto il modulo **alchemist-loading** del simulatore. Gli interventi eseguiti in classi al di fuori di questo modulo riguardano l'adattamento delle classi e dei test alla nuova architettura.

3.4.1 Strumenti di sviluppo

Git

Git[2] è un sistema di controllo di versione distribuito (DVCS) che permette di tenere traccia delle modifiche effettuate durante lo sviluppo di software. Tra le varie funzionalità, git offre la possibilità di mantenere due versioni dello stesso progetto: una stabile e una in via di sviluppo; tale organizzazione risulta piuttosto flessibile per fronteggiare i problemi che solitamente si incontrano in fase di implementazione. Alchemist è sviluppato con la metodologia di lavoro nota come Gitflow[3], che prevede che ogni componente del team di sviluppo abbia una propria copia del progetto principale: l'aggiunta di codice al repository centrale avviene per mezzo di *pull requests*, con le quali un membro del team richiede che il suo codice sia aggiunto a tale repository. Il codice è stato quindi sviluppato in una copia del repository originale e sotto il branch *develop*. Il codice inerente allo sviluppo di un esportatore su MongoDB è stato scritto sotto il branch *feature-mongo*.

[1] `https://bsonspec.org/spec.html`
[2] `https://git-scm.com/`
[3] `https://nvie.com/posts/a-successful-git-branching-model/`

Gradle

Gradle[4] è uno strumento per l'automazione della costruzione del software, organizzato in task per semplificare la compilazione del codice sorgente, la risoluzione delle dipendenze, l'esecuzione dei test automatici e il controllo di qualità del codice. In breve, esso è in grado di determinare quali porzioni del sistema software necessitano di essere ricostruite riducendo significativamente il tempo di costruzione del progetto. Questo sistema permette di avere una gestione controllata delle dipendenze le quali vengono scaricate dai vari repository Maven[5] durante la fase di compilazione. Il simulatore Alchemist, mantiene, oltre a quello generale di progetto, uno script Gradle per ogni modulo presente al suo interno, nel quale sono specificati gli altri moduli o eventuali librerie esterne di cui si vogliono utilizzare le funzionalità.

GitHub CI

Il simulatore utilizza un sistema di *Continuous Integration* per monitorare il comportamento del software allo stato attuale, installandolo ed eseguendolo su diverse macchine virtuali, con diversi sistemi operativi e diverse configurazioni. Questa possibilità, permette di tenere sempre sotto controllo la retrocompatibilità ed il supporto multi-piattaforma del simulatore, testandone il funzionamento con diverse versioni della JVM su tutti i principali sistemi operativi. L'intero processo è configurabile attraverso un file YAML e viene eseguito ad ogni push sul repository.

Kotlin

Tutte le classi sviluppate sono state scritte in Kotlin[6], un linguaggio di programmazione open-source. La sua principale caratteristica è quella di essere pienamente interoperabile con il linguaggio Java e la Java Virtual Machine (JVM) e cioè di poter sfruttare tutte le librerie presenti per Java e al contempo di poter essere importato all'interno di codice Java senza riscontrare particolari problemi. Kotlin è un linguaggio multi-paradigma, in quanto presenta sia caratteristiche tipiche della programmazione orientata agli oggetti sia del paradigma funzionale, e multi-piattaforma, poiché, oltre che per la JVM, può essere compilato per Android, iOS, JavaScript e nativamente per Windows, Linux e MacOS. La scelta dell'uso di Kotlin al posto di Java per l'implementazione degli esportatori all'interno di Alchemist, si è rilevata molto utile in diversi frangenti, nei quali altrimenti, per ottenere lo stesso risultato, si sarebbe dovuto adottare soluzioni molto più verbose

[4] `https://gradle.org/`
[5] `https://search.maven.org/`
[6] `https://kotlinlang.org`

e di difficile comprensione. Un esempio di ciò lo si può notare nei costruttori degli esportatori. Grazie all'utlizzo della notazione `@JVMOverloads` verrà generata, per ogni parametro di default, l'overload della funzione. Per ottenere lo stesso risultato in Java sarebbe stato necessario creare molteplici costruttori, in particolare uno per ciascuna combinazione di parametri.

Librerie esterne

Durante lo sviluppo del progetto, in particolare nell'implementazione dell'esportatore per il database MongoDB è stata utilizzata la libreria **mongo-driver-sync**[7]. La libreria contiene i driver del database Mongo per il linguaggio Java ossia un insieme di metodi per aiutare lo sviluppatore a connettersi ed interagire con il database. Gli oggetti e le strutture dati fornite dalla libreria, come ad esempio *MongoClient*, *MongoCollections* e *MongoDatabase* sono risultati estramamente utili in termini di efficienza e pulizia del codice.

Valutazione del codice

Date le dimensioni di Alchemist, è necessario fare uso di strumenti che controllino la qualità del codice e diano la possibilità di testarlo in modo immediato. Gli strumenti di qualità del codice permettono di revisionare il codice in modo sistematico, così da evitare errori che a volte possono verificarsi, senza bisogno che il programma venga realmente eseguito: essi analizzano il codice sorgente per individuare potenziali bug o codice duplicato e per indicare i possibili miglioramenti e ottimizzazioni. Alchemist utilizza i seguenti strumenti:

- **checkstyle**: è uno strumento open-source per diversi linguaggi di programmazione e verifica che il codice scritto aderisca a un determinato stile di codifica. Esso effettua un controllo sulla presentazione e non sul contenuto, dunque non permette di assicurare la correttezza e la completezza del software.

- **detekt**: è uno strumento open-source di analisi del codice per il linguaggio Kotlin.Prevede un set di regole, personalizzabile dallo sviluppatore, per definire quando una parte del codice è errata.

- **pmd**: è uno strumento di analisi del codice sorgente per Java e altri linguaggi. Generalmente, gli errori segnalati riguardano scelte implementative subottimali e imperfezioni nel codice, che non inficiano direttamente il funzionamento del programma ma per la maggior parte il livello della qualità del sorgente.

[7] `https://search.maven.org/artifact/org.mongodb/mongodb-driver-sync/3.10.1/jar`

- **ktlint**: è uno strumento di analisi del codice sorgente per il linguaggio Kotlin. Contiene delle regole standard ma offre anche la possibilità di aggiungere un file di configurazione contente le regole da rispettare.
- **spotbugs**: è uno strumento open-source che opera direttamente sul bytecode Java e rileva possibili bug all'interno del codice. Classifica i potenziali errori in categorie, per dare un'idea migliore allo sviluppatore di quale potrebbe essere il loro impatto sul software.
- **cpd**: è un plugin che analizza il codice sorgente con lo scopo di ricerca di codice duplicato.

Ambiente di sviluppo integrato

Un IDE (Integrated Development Environment), o ambiente di sviluppo integrato, è un software che aiuta i programmatori nello sviluppo del codice sorgente di un programma, mettendo a disposizione una serie di strumenti che permettono scrittura, compilazione o interpretazione, debug e analisi del codice da un unico ambiente, appunto, integrato. Durante lo sviluppo è stato utilizzato l'ambiente di sviluppo **IntelliJ IDEA**[8], sviluppato da JetBrains, nell'edizione *ultimate*.

3.4.2 Testing

In parallelo alla fase di sviluppo è stata portata avanti un'attività di Unit testing per ogni nuova funzionalità introdotta, con lo scopo di verificarne il corretto funzionamento e impedendo così che nuove modifiche introdotte nel simulatore possano compromettere il suo stato di avanzamento attuale. Per la creazione dei test sono state utilizzate le librerie *Kotest*[9] e *JUnit*[10], grazie alle quali è stato possibile testare dei comportamenti in un linguaggio simile al naturale. Ciascun test effettuato è stato affiancato da un opportuno file di configurazione in formato YAML. In questo modo è stato possibile lanciare simulazioni e testare le varie fasi di interesse. In particolare, all'interno della classe **TestExportersCreation** a partire da un file di configurazione in cui venivano dichiarati molteplici esportatori, è stato testato il corretto caricamento degli stessi e per ciascuno di essi sono state testate le strutture dati necessarie. Un'altra funzionalità della quale è necessario verificare la corretta esecuzione è l'esportazione su file. Partendo da un file di configurazione in cui veniva dichiarato un *CSVExporter* nella classe **TestCSVExporter** è stata testata la corretta appartenenza alla classe e la presenza del file di output generato.

[8] `https://www.jetbrains.com/idea/`
[9] `https://kotest.io/`
[10] `https://junit.org/junit5/`

Files					Complexity	Coverage
MongoService.kt	14	9	5	0	54.55%	64.29%
GlobalExporter.kt	13	13	0	0	100.00%	100.00%
AbstractExporter.kt	14	14	0	0	85.71%	100.00%
MongoDBExporter.kt	26	25	1	0	83.33%	96.15%
CSVExporter.kt	53	28	24	1	22.58%	52.83%
Folder totals (5 files)	120	89	30	1	50.77%	74.17%
Project totals (565 files)	17448	6172	761	10515	31.58%	35.37%

Figura 3.8: Coverage del codice sviluppato.

La nuova architettura permette di specificare intervalli di campionamento differenti a seconda della tipologia di esportatore. Questa funzionalità è stata testata all'interno della classe **TestExportInterval**. Per quanto riguarda l'esportazione su database è necessario testare inizialmente se è presente un'istanza attiva del database. Nello specifico, nella classe **TestMongoInstance** viene testato il corretto funzionamento della libreria `embedded-mongodb` ossia se è presente un processo di mongodb in locale e in caso affermativo se è possibile connettersi ed interagire con il database. Per quanto riguarda invece l'esportatore *MongoDBExporter*, nella classe **TestMongoDBExporter** viene testata la connessione al database MongoDB con i parametri forniti nel file di configurazione. Inoltre è stata testata la creazione del database e l'effettiva presenza dei dati esportati in esso.

Capitolo 4

Conclusioni e sviluppi futuri

4.1 Conclusioni

Lo scopo di questo lavoro era quello di analizzare il dominio della generazione di dati tramite simulazione, analizzare lo stato dell'arte e infine applicare quanto appreso nella ri-progettazione di un modulo di esportazione dati per il simulatore Alchemist. A lavoro terminato possiamo affermare che l'architettura precedente ha subito notevoli modifiche. Tra i cambiamenti più rilevanti troviamo:

- possibilità di scegliere la tipologia di esportazione direttamente nel file di configurazione;
- eliminazione dei parametri *export* ed *interval*: l'utente non dovrà più specificare il nome del file e l'intervallo di campionamento come parametri della simulazione ma verranno indicati opportunamente nel file di configurazione;
- creazione di un'interfaccia di export e di una classe astratta contenente i metodi comuni a tutti gli esportatori;
- sviluppo di un esportatore verso il database *MongoDB*.

4.2 Sviluppi futuri

Una delle principali caratteristiche del nuovo sistema di export riguarda sicuramente la flessibilità e la dinamicità con cui possono essere aggiunti nuovi esportatori. In particolare, sarà necessario solamente estendere la classe `AbstractExporter` ed implementare il metodo principale per esportare i dati. Una volta creata basterà aggiungere il nome della nuova classe dell'esportatore nel file di configurazione e il sistema lo aggiungerà in automatico alla lista di esportatori presenti. Il lavoro

potrà quindi essere esteso aggiungendo delle nuove tipologie di esportatori come ad esempio un'implementazione per un differente database oppure un esportatore responsabile di creare grafici in tempo reale con i dati generati dalla simulazione.

Bibliografia

[1] Khaleel Ahmad and Mohd Javed. *Hands-On Redis*, pages 355–364. 03 2017.

[2] Miroslav Babiš and Peter Magula. Netlogo — an alternative way of simulating mobile ad hoc networks. In *2012 5th Joint IFIP Wireless and Mobile Networking Conference (WMNC)*, pages 122–125, 2012.

[3] Ganesha Beedubail and Udo Pooch. An architecture for object replication in distributed systems. 04 1996.

[4] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML ain't markup language (YAML) (tm) version 1.2. Technical report, YAML.org, 9 2009.

[5] T. Blochwitz, M. Otter, Johan Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, Andreas Junghanns, Jakob Mauss, D. Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. *9th International Modelica Conference*, pages 173–184, 01 2012.

[6] Joe Celko. *Databases Versus File Systems*, pages 1–10. 12 2011.

[7] P.A. Fishwick. Computer simulation. *IEEE Potentials*, 15(1):24–27, 1996.

[8] Joés M. Garrido. *Simulation Output Analysis*, pages 405–409. Springer US, Boston, MA, 2009.

[9] Harvey Gould, Jan Tobochnik, Wolfgang Christian, and Eric Ayars. An introduction to computer simulation methods: Applications to physical systems, 2nd edition. *American Journal of Physics - AMER J PHYS*, 74:652–653, 01 2006.

[10] Guy Harrison and Michael Harrison. *MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications*. 01 2021.

[11] Guy Harrison and Michael Harrison. *MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications*. 01 2021.

[12] George Heineman and Jeremy Denham. Entity, boundary, control as modularity force multiplier. 01 2009.

[13] A.R. Ashok Kumar, S.V. Rao, and Diganta Goswami. Ns3 simulator for a study of data center networks. In *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, pages 224–231, 2013.

[14] Pramod Kumbhar, Michael Hines, Jeremy Fouriaux, Aleksandr Ovcharenko, James King, Fabien Delalondre, and Felix Schürmann. Coreneuron : An optimized compute engine for the neuron simulator, 2019.

[15] Liu Li-li and Ji Chang-peng. Applications and studies of the computer simulation methods in physics. In *2009 4th International Conference on Computer Science Education*, pages 1368–1371, 2009.

[16] Yingshu Liu and Cheng Yang. Omnet++ based modeling and simulation of the ieee 1588 ptp clock. In *2011 International Conference on Electrical and Control Engineering*, pages 4602–4605, 2011.

[17] Michael North, Nicholson Collier, J. Ozik, Eric Tatara, C. Macal, M. Bragen, and Pamela Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1, 10 2013.

[18] Vaskaran Sarcar. *MVC Pattern*, pages 495–519. 09 2020.

[19] Andrea Unger, Susanne Biermann, Mathias John, Adelinde Uhrmacher, and Heidrun Schumann. Visual support for modeling and simulation of cell biological systems. In *2007 Winter Simulation Conference*, pages 2378–2378, 2007.

[20] Joseph M. Whitmeyer, Moutaz Khouja, Ted Carmichael, Amar Saric, Chris Eichelberger, Min Sun, and Mirsad Hadzikadic. A computer simulation laboratory for social theories. In *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 2, pages 512–515, 2008.

[21] Eric Winsberg. Computer Simulations in Science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2019 edition, 2019.

[22] Tom Woo. How do we simulate things at the scale of molecules and electrons? an introduction to the technology and hpc aspects of computational chemistry. In *2008 22nd International Symposium on High Performance Computing Systems and Applications*, pages 3–3, 2008.

[23] Wang Xiang-Hui and Li Zeng-Xin. The economic model optimization and application based on computer simulation technology. In *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, pages 956–959, 2015.

[24] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation—Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, Inc., San Diego, CA, 2nd edition, 2000.

Ringraziamenti

Un sincero ringraziamento va a tutti coloro che mi hanno aiutato in vario modo a raggiungere questo traguardo.
Ringrazio tutti i professori del corso di studi in Ingegneria e Scienze Informatiche, che mi hanno trasmesso grande passione per le loro materie.
In particolar modo, ringrazio il professor Danilo Pianini e il professor Mirko Viroli per l'opportunità che mi hanno concesso e per le numerose nozioni apprese durante il progetto.
Ringrazio i miei genitori ed i miei amici, che mi hanno supportato incondizionatamente durante il percorso.
Ringrazio infine i miei compagni di studio, con i quali ho condiviso tre bellissimi anni e che sono stati una spalla su cui contare dall'inizio alla fine della mia carriera universitaria.

Appendice A

Esempi di file di simulazioni

A.1 Sezione di export con CSVExporter

```
export:
  - type: CSVExporter
    parameters:
      filename: "00-testing_csv_export"
      appendTime: true
      interval: 2.0
    data:
      - time
      - molecule: "default_module:default_program"
        aggregators: [ mean, max, min, variance, median ]
        value-filter: onlyfinite
```

A.2 Sezione di export con MongoDBExporter

```
export:
  - type: MongoDBExporter
    parameters:
      uri: "mongodb+srv://alchemist-user:alchemist-pwd
           @alchemist.ewxds.mongodb.net"
      appendTime: false
      dbname: "alchemist-mongo-test"
      interval: 3.0
    data:
      - time
```

```
      - molecule: "default_module:default_program"
        value-filter: onlyfinite
```

A.3 Sezione di export con più esportatori

```
export:
 - type: CSVExporter
   parameters:
     filename: "00-testing_csv_export"
     appendTime: true
     interval: 2.0
   data:
     - time

 - type: MongoDBExporter
   parameters:
     uri: "mongodb+srv://alchemist-user:alchemist-pwd
          @alchemist.ewxds.mongodb.net"
     appendTime: false
     dbname: "alchemist-mongo-test"
     interval: 3.0
   data:
     - time
     - molecule: "default_module:default_program"
```

A.4 Simulazione di esempio completa

```
 incarnation: protelis
variables:
  zoom: &zoom
    formula: 0.1
  image_name: { formula: "'chiaravalle.png'" }
  image_path: &image_path
    language: kotlin
    formula: >
      import java.io.File
      File("../..").walkTopDown().find {
      image_name in it.name
```

```
      }?.absolutePath ?: image_name
  walking_speed: &walk-speed {
    default: 1.4, min: 1, max: 2, step: 0.1 }
  seed: &seed { default: 0, min: 0, max: 99, step: 1 }
  scenario_seed: &scenario_seed { formula: (seed + 31) * seed }
  people_count: &people_count
    type: GeometricVariable
    parameters: [300, 50, 500, 9]
seeds: { simulation: *seed, scenario: *scenario_seed}
export:
  - type: CSVExporter
    parameters:
      filename: "00-testing_csv_export"
      appendTime: false
      interval: 2
    data:
      - time
        aggregators: [ mean, max, min, variance, median ]

  - type: MongoDBExporter
    parameters:
      uri: "mongodb+srv://alchemist-user:alchemist-pwd
            @alchemist.ewxds.mongodb.net"
      dbname: "alchemist-testing-mongo"
      interval: 3.0
    data:
      - molecule: "default_module:default_program"
        value-filter: onlyfinite

environment: { type: ImageEnvironment,
               parameters: [*image_path, *zoom] }
network-model: { type: ObstaclesBreakConnection,
                 parameters: [50] }
deployments:
  type: Rectangle
  parameters: [*people_count, 62, 15, 95, 200]
  programs:
    - time-distribution: 1
      program: >
        import protelis:coord:spreading
```

```
        let source = [110, 325]
        let vector = self.getCoordinates() - source
        let distance = hypot(vector.get(0), vector.get(1))
        distanceTo(distance < 50)
    - program: send
    - { type: Event, time-distribution: 1, actions: {
        type: LevyWalk, parameters: [*walk-speed] } }

terminate:
  - type: AfterTime
    parameters: 5
```