Implementazione di una libreria per la Simulazione a Eventi

Foni Gianmarco
s347952

1 Introduzione

Questa libreria vuole essere un supporto per la simulazione di semplici modelli a eventi discreti. L'idea principale è separare la logica di scheduling dagli oggetti simulati, tramite classi dedicate per risorse, code e gestori di eventi.

2 Struttura della libreria

La struttura della libreria si ispira, almeno in parte, a quella in [1], con alcune ulteriori considerazioni semplificative, che chiariscono il contesto in cui può essere usata.

Effettuare una simulazione significa osservare la realizzazione di eventi che accadono in sequenza secondo una lista di eventi. Questa lista si aggiorna dinamicamente durante l'esecuzione: risolvere un evento può implicare la creazione e la schedulazione di nuovi eventi nella lista.
Gli oggetti del modello sono rappresentati tramite *Entità*, in generale, un evento ha l'effetto di modificare lo stato e le caratteristiche di una o più entità a cui è associato. Queste possono avere oggetti come code o magazzini di risorse e interagiscono fra loro, muovendosi lungo una rete di code o consumando risorse secondo le regole del modello.

Le classi di questa libreria sono organizzate come in figura 1. La classe *SimulatorManager* è quella centrale che costruisce e avvia la simulazione, prendendo informazioni da *EventsList* e da *ScenarioGenerator* e restituendo un file report con i risultati di simulazione calcolati da *Statistics*. Il metodo *StartSimulation* contiene il motore principale di simulazione: estrae il primo evento disponibile in coda *EventsList.eventQueue*, lo esegue e ne analizza gli effetti. Ogni oggetto *Events* contiene un'indicazione sull'entità che dovrà eseguirlo. In questa libreria l'esecuzione è una responsabilità delegata dall'evento all'entità (*Entity*) colpita, che deve avere un metodo di gestione implementato specifico per quell'evento. Per una maggiore fruibilità della libreria, non è necessario

derivare in ogni modello una sottoclasse o modificare i metodi della classe Entity, è possibile utilizzare delle funzioni di gestione eventi, definite in locale e passate alla classe entity come proprietà *eventHandlers*.
Il risultato è che ogni entità avrà un set di funzioni con cui può gestire eventi nei casi specifici di modello. *Entity.handleEvent()* si occupa di cercare il metodo o funzione handle con cui gestire l'evento. Questa dinamica di gestione, sebbene risulti costosa per via delle ripetute chiamate di funzione, permette di utilizzare la libreria in contesti differenti senza dover implementare nuovi metodi specifici (questo resta comunque consentito e potrebbe essere importante nell'ottica di operare una tipizzazione degli eventi).

Le entità hanno associate alcune proprietà: *state* è lo stato dell'entità e contiene quelle informazioni che possono variare nel corso della simulazione, per effetto degli eventi. *info* è invece una *struct* che contiene i parametri di configurazione correnti, questi parametri definiscono lo scenario di simulazione e vengono aggiornati da *Entity.setConfig(newParams)* su chiamata del metodo "maestro" *SimulatorManager.setConfig(newParams)*, ogni volta che si passa ad un nuovo scenario. Distinguiamo due tipi di entità, in base alle caratteristiche che hanno associate: le *TransactionBasedEntity* rappresentano un oggetto mobile o un flusso che attraversa il sistema e interagisce con le risorse. Gestiscono una coda di oggetti o informazioni, e implementano metodi specifici per la loro gestione. Le *ResourceBasedEntity* invece rappresentano una risorsa limitata (e.g. un server, una macchina o un operatore) con una certa capacità disponibile.

Per poter svolgere un'analisi statisticamente significativa, è possibile indicare in *ScenarioGenerator* un set di parametri di simulazione, ad uno o più livelli, (*ScenarioGenerator.values* contiene questa informazione). Questa classe si occupa di costruire un piano degli esperimenti, provando le varie combinazioni di livello dei parametri, ognuna di queste configurazioni definisce uno "scenario". La classe imposta i limiti di simulazione *timeHorizon* nel motore di simulazione e fornisce i parametri di scenario al simulatore, tramite la chiamata di *obtainScenarios*.

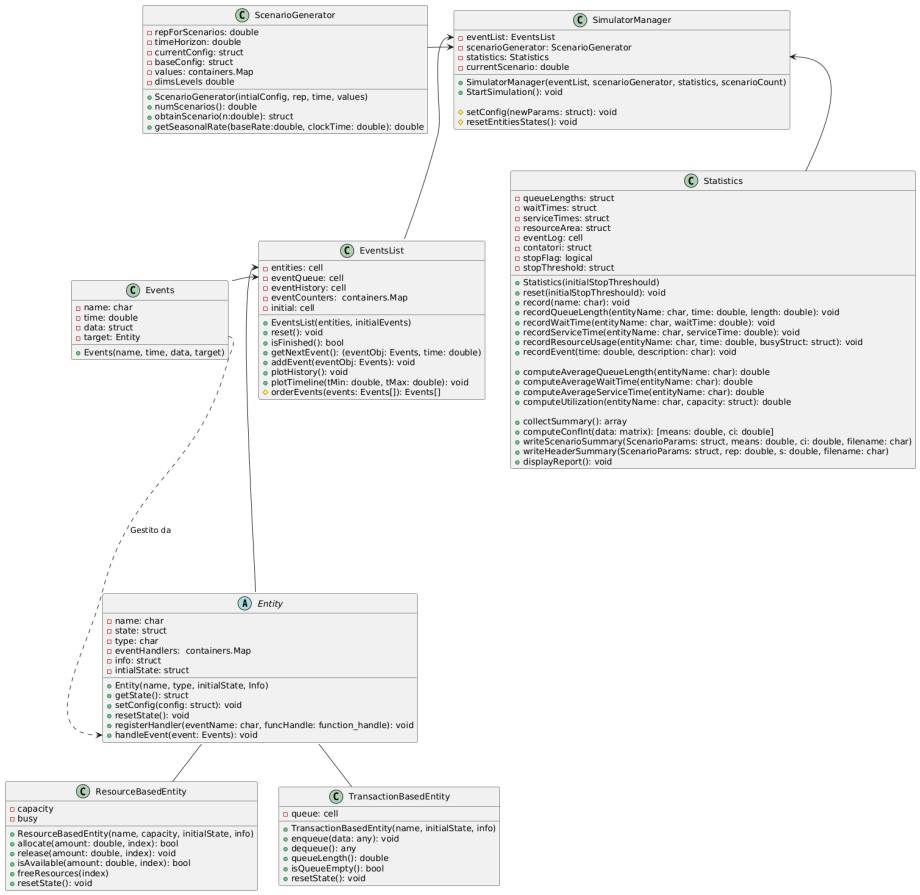

Figure 1: Organizzazione delle classi della libreria

3 Esperimenti

La libreria è utilizzata e testata in tre contesti differenti, di complessità crescente, che catturano e sfruttano la generalità delle classi:

- Caso di coda semplice $M/M/1$, (file: *main_MM1.m*) utile per poter confrontare i risultati ottenuti con quelli teorici.
- Modello del venditore di panini, (coda panini con buffer e coda clienti, file: *main_Roll.m*) in cui i risultati sono stati confrontati con la versione script *rollQueue.m* per verifica.
- Modello del benzinaio, file *main_Benzinaio.m*, in questo caso, il corretto funzionamento della simulazione è stato testato nel file *test_benzinaio.m*, di seguito presentiamo l'impostazione e i risultati di simulazione (1).

4 Simulazione 'Benzinaio'

Per simulare il modello del benzinaio sono stati costruiti i seguenti oggetti:

- 1 Entità TransactionBased *VehicleQueue* con una coda associata per gestire l'ingresso delle macchine nella stazione di servizio, le macchine arrivano secondo un Processo di Poisson e verificano se ci sono posti disponibili guardando la lunghezza della coda.
- 1 Entità ResourceBased *Pumps* per modellare le pompe di servizio, l'entità gestisce i 4 posti disponibili, rappresentati in *entity.capacity* come *struct('A',1, 'B',1, 'C',1, 'D',1)*, allocando ogni macchina alla pompa corrispondente e tenendo sotto controllo il livello di utilizzo.
- 1 Entità TransactionBased *Cashier* che rappresenta il pagamento in cassa. A questa entità è associata una coda, in cui i clienti si aggiungono completato il servizio (quindi dopo un evento di tipo *RefuelOn*), e lasciano soltanto dopo aver aspettato il tempo di pagamento ed essersi assicurati di avere una via di uscita libera.

Gli eventi che si susseguono sono:

- Evento *Arrival* (gestito da VehicleQueue), rappresenta l'arrivo di una nuova macchina nel sistema, questa entra nella stazione di servizio solo se ci sono dei posti in coda liberi, altrimenti è conteggiata come *lost*. Se sono il primo in coda e ci sono pompe libere, VehicleQueue mi assegna ad una pompa generando un evento *RefuelOn* e smaltendo quindi la coda.
- Eventi *RefuelOnA*, *RefuelOnB*, *RefuelOnC*, *RefuelOnD*. Quando si libera una postazione e facciamo scorrere la coda di VehicleQueue viene generato uno di questi 4 eventi, a seconda della pompa disponibile e compatibile. Questi eventi, che rappresentano il tempo di completamento del rifornimeto, sono gestiti da Pumps. Idealmente in questo istante il cliente ha terminato di fare benzine e, lasciando il posto auto occupato, si dirige in cassa per effettuare il pagamento. Pumps inserisce quindi il cliente nella coda di Cashier, e se il cassiere è libero (nessun altro in coda), genera un evento di pagamento.
- Evento *PaymentCompleted*, questo evento gestito da Cashier, corrisponde al termine del pagamento in cassa di un cliente, notiamo che ciò non corrisponde necessariamente all'uscita del cliente dal sistema (vedi fig.2). A questo punto, se la macchina che ha pagato può uscire, lo fa, liberando il proprio posto ed eventualmente anche il posto precedente. Ogni volta che si libera una pompa la coda di VehicleQueue viene fatta scorrere di conseguenza, riassegnando soltanto le pompe liberate e accessibili.

4.1 Test e validazione

In ottica di validare la simulazione sono stati svolti alcuni semplici test, sicuramente non esaustivi, ma comunque sufficienti a dare un idea di massima del

funzionamento.
I test sono raccolti in *test_benzinaio.m*, (attenzione alle parti commentate), comprendono l'analisi di semplici situazioni in cui:

- Il sistema è fermo e si iniettano artificialmente dei veicoli asserendo la sequenza di eventi che deve essere generata.
- Si usa un sistema ridotto con una sola pompa e si confrontano i risultati con quelli teorici di un sistema $M/G/1$

Il file comprende anche il codice per l'output grafico in figura 2, usato anch'esso per analizzare ciò che accade simulando. Dal grafico si vede come vengono gestiti alcuni casi patologici:

- il cliente 7 entra nel sistema al tempo 20.21 e ha bocchetta lato destro, quando entra però deve attendere in coda perché una macchina in A gli stà impedendo di accedere alla pompa B che invece sarebbe libera. Solo dopo che A ha completato il servizio e ha pagato (uscendo), il cliente 7 può accedere alla pompa B (ora libera e anche accessibile).
- Subito dopo il cliente 7 sono arrivati alla stazione di servizio il cliente 6 (al tempo 21.91) e il cliente 8 al tempo 21.92, entrambi attendono in coda occupando gli ultimi due posti dietro al cliente 6. In blu sono segnalati altri clienti che trovano la coda piena e non entrano. Nell'istante in cui il cliente 5 esce dal sistema liberando l'accesso al cliente 6, il 7 e 8 accedono alle pompe in contemporanea.
- Il cliente 13 mostra il caso in cui un cliente, dopo aver completato il rifornimento e aver effettuato il pagamento, non può comunque uscire dal sistema, trovando occupata la pompa di fronte a lui. Il cliente 13 che occupa la pompa A è costretto ad attendere in cassa un tempo ulteriore finché anche il cliente 14 nella pompa B completa il pagamento. Escono poi allo stesso istante dal sistema liberando solo allora le due pompe.

References

[1] Jeffrey A. Joines and Stephen D. Roberts. Object-oriented simulation. In Jerry Banks, editor, *The Handbook of Simulation*, chapter 11, pages 437–459. Wiley, New York, 1998.

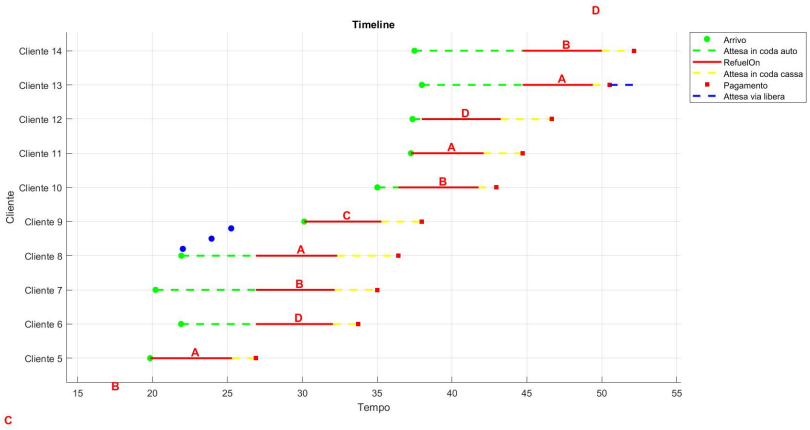

Figure 2: Esempio di simulazione con: Processo di Poisson degli arrivi in coda di intertempo 4 min, tempi di servizio in pompa distribuiti uniformemente in [4.5, 5.5] e tempi di servizio in cassa uniformemente distribuiti in [1, 2], l'unità tempora il minuto.

Parametri	**Scenario 1**	**Scenario 2**	**Scenario 3**	**Scenario 4**	**Scenario 5**
servRatePump	3.5	3.5	5.0	7.0	7.0
servRate	2.0	2.0	2.0	2.0	2.0
ArrivalRate	4.0	4.0	4.0	4.0	6.0
Stagionalita	0.2	0.5	0.2	0.2	0.2
maxPlaces	3.0	3.0	3.0	3.0	3.0
Statistiche	**Scenario 1**	**Scenario 2**	**Scenario 3**	**Scenario 4**	**Scenario 5**
toServe	327.960 ± 2.823	324.760 ± 3.607	303.710 ± 2.622	263.650 ± 2.825	211.150 ± 2.307
lost	32.650 ± 1.862	35.160 ± 2.346	55.820 ± 2.903	94.210 ± 3.727	26.510 ± 2.008
AvgQueueLen_vehicleQueue	0.544 ± 0.016	0.551 ± 0.019	0.813 ± 0.023	1.193 ± 0.028	0.610 ± 0.029
AvgQueueLen_cashier	0.640 ± 0.010	0.635 ± 0.012	0.564 ± 0.009	0.481 ± 0.010	0.368 ± 0.008
AvgWait_vehicleQueue	2.374 ± 0.069	2.423 ± 0.080	3.820 ± 0.107	6.472 ± 0.188	4.109 ± 0.189
AvgWait_cashier	2.806 ± 0.038	2.812 ± 0.043	2.670 ± 0.040	2.618 ± 0.047	2.505 ± 0.044
Util_A	0.330 ± 0.010	0.327 ± 0.010	0.412 ± 0.010	0.475 ± 0.012	0.335 ± 0.012
Util_B	0.457 ± 0.007	0.447 ± 0.008	0.500 ± 0.008	0.520 ± 0.009	0.454 ± 0.009
Util_C	0.330 ± 0.009	0.329 ± 0.009	0.397 ± 0.011	0.492 ± 0.013	0.327 ± 0.012
Util_D	0.454 ± 0.007	0.452 ± 0.006	0.494 ± 0.008	0.531 ± 0.009	0.448 ± 0.008

Table 1: Esempio di statistiche, estratte da *simulation_report.txt*, durante vari scenari usando tempi sempre generati da distribuzioni esponenziali. Gli intervalli di confidenza sono calcolati ripetendo 100 replicazioni per ogni scenario e sono a livello 95%. A discapito del nome, *ServRate*, *ServRatePump* e *ArrivalRate*, rappresentano le medie degli esponenziali, non i tassi. L'unità di tempo è il minuto. Ogni replicazione ha un orizzonte temporale di 1440 minuti per simulare le 24h della giornate. Il parametro di stagionalità è un fattore che cambia ArrivalRate, in base al tempo della giornata, in modo che gli arrivi abbiano un picco alle ore 12a.m. e un minimo alle 12p.m.