POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Ambiente di simulazione per processi di business B2B

Relatore
prof. Giorgio Bruno

firma del relatore
.....

Candidato
Aldo Jaku
firma del candidato
.....

A.A. 2017/2018

Indice

- **Capitolo 6: L’ambiente di simulazione**
  - 6.1 Introduzione
  - 6.2 La classe Net
  - 6.3 La classe P
  - 6.4 La classe Entity
  - 6.5 La classe T
  - 6.6 La classe TCode
  - 6.7 La classe TComparator
  - 6.8 La classe Event
  - 6.9 La classe EventComparator
  - 6.10 La classe Simulator
- **Capitolo 7: Esempio semplice**
  - 7.1 Production System 1
    - 7.1.1 Process model
    - 7.1.2 Modello informativo
    - 7.1.3 Applicazione dell’ambiente
    - 7.1.4 Risultati ottenuti
- **Capitolo 8: Esempi con rete di Petri Avanzata**
  - 8.1 Production System 2
    - 8.1.1 Process model
    - 8.1.2 Modello informativo
    - 8.1.3 Applicazione dell’ambiente
    - 8.1.4 Risultati ottenuti
  - 8.2 Supermarket
    - 8.2.1 Process model
    - 8.2.2 Modello informativo
    - 8.2.3 Applicazione dell’ambiente
    - 8.2.4 Risultati ottenuti
- **Capitolo 9: Esempio con processo B2B**
  - 9.1 Buyer-Seller
    - 9.1.1 Modello collaborativo
    - 9.1.2 Process Model
    - 9.1.3 Modello informativo
    - 9.1.4 Applicazione dell’ambiente
    - 8.1.5 Risultati ottenuti
- **Capitolo 10: Conclusione**
    - 10.1 Possibili sviluppi futuri

Capitolo 1

Simulazione a eventi discreti

1.1 Introduzione

Una simulazione a eventi discreti considera il sistema come una sequenza di eventi discreti nel tempo: ogni evento si verifica in un particolare istante t e attua un cambiamento di stato nel sistema. Quindi tra due eventi consecutivi, si assume che non ci sia alcun cambiamento di stato e pertanto una volta concluso l'evento in un certo istante t_1 si può passare direttamente all'istante di inizio dell'evento successivo t_2.

Essa si contrappone alla simulazione continua in cui, invece, lo stato è una funzione continua nel tempo e può variare in qualunque istante t; di conseguenza la simulazione deve tracciare la variazione dello stato in modo continuo. Infatti, in questo caso, non si parla di simulazione event-based (come nel caso della simulazione a eventi discreti), ma di simulazione activity-based: il tempo viene suddiviso in n piccoli intervalli e lo stato viene aggiornato in base alle attività avvenute nei singoli intervalli comportando un aumento di complessità della simulazione. Per questo motivo, le simulazioni continue sono solitamente più lente rispetto a quelle a eventi discreti.

Queste ultime seguono un metodo più semplice, ne è un esempio il metodo a 3 fasi: nella prima si passa all'evento cronologicamente successivo; nella seconda vengono eseguiti tutti gli eventi senza condizioni che avvengono in quell'instante (essi vengono chiamati B-Events); nella terza vengono eseguiti tutti gli eventi che avvengono sotto certe condizioni in quell'istante (detti anche C-Events). Questo metodo espande il caso generale di approccio event-based, in cui gli eventi vengono ordinati in modo da sfruttare in modo efficiente le risorse computazionali a disposizione. L'approccio a 3 fasi viene utilizzato in alcuni pacchetti software commerciali, anche se il metodo di simulazione resta solitamente nascosto all'utente finale.

Un esempio di simulazione a eventi discreti è dato da una coda di clienti che arrivano alla cassa e devono essere serviti da uno o più commessi. In questo caso le entità del sistema sono i clienti in coda e i commessi. Gli eventi sono solo cliente-arriva e cliente-esce perché possiamo includere l'evento commesso-serve-cliente nella logica degli altri due. Essi attuano un cambiamento sugli stati del sistema, in questo caso possiamo considerare come stati: il numero di clienti in coda, rappresentato da un intero da 0 a n e lo stato del commesso , rappresentato da una variabile booleana che indica se è libero o occupato. La simulazione ha inoltre bisogno di variabili casuali: la

prima rappresenta ogni quanto arriva un nuovo cliente e la seconda il tempo necessario al commesso per servire un cliente.

1.2 Le componenti

La struttura del programma è composta da 3 livelli di software:

- Il software esecutivo della simulazione: detto anche simulation control program o simulator, controlla l'esecuzione del model program, mette in sequenza le operazioni e gestisce i problemi legati alla modularità separando il controllo generico dai dettagli del modello. Sono possibili due tecniche: la tecnica ad esecuzione sincrona e la tecnica dell'event-scanning. L'algoritmo generico della tecnica ad esecuzione sincrona è:

```
while (la simulazione non è terminata){
   CurrentTime +=valorePredefinito;
   if (si verificano eventi in questo intervallo)
       simulate (event);

}
```

Questo algoritmo ha due limiti: gli eventi vengono eseguiti alla fine dell'intervallo temporale invece che in un istante di tempo preciso; alcuni intervalli potrebbero non avere eventi e sprecare quindi cicli di CPU. L'algoritmo generico della seconda tecnica, invece, è il seguente:

```
while (ci sono eventi nella event list && la simulazione non è terminata){

  prendi dalla coda l'evento a tempo minore non ancora simulato;
  currentTime=eventTime;
  simulate (event);

}
```

Che non ha più i limiti della prima tecnica, in questo caso la simulazione viene detta event-driven perché il tempo avanza ad ogni giro al valore del primo evento della event list. Entrambe le tecniche sono in grado di simulare il parallelismo tra i processi.

- Il model program: un programma che implementa il modello del sistema da simulare, ossia il simulation model. Viene messo in esecuzione dal simulator.
- Gli strumentidi routines: per generare numeri casuali derivandoli da tipiche distribuzioni di probabilità, per ricavare statistiche e per gestire i problemi legati alla modularità separando le funzioni generiche da quelle specifiche del modello.

Oltre alla logica di funzionamento, un sistema di simulazione event-based ha bisogno di componenti che descrivono le caratteristiche del sistema studiato:

- Lo stato del sistema: è costituito da un insieme di variabili che rappresentano le proprietà più importanti del sistema studiato. Lo stato ha una sua traiettoria nel tempo S(t) che viene rappresentata matematicamente da una funzione a tratti il cui valore cambia ogni volta che si verifica un evento.
- Il clock: la simulazione deve tenere traccia del tempo corrente, in qualunque unità di misura purchè adatta al sistema studiato. Il tempo nelle simulazioni a eventi discreti, diversamente da quelle a tempo continuo, salta da un istante all'altro, poiché si assume che gli eventi siano istantanei; pertanto il clock viene aggiornato di volta in volta, passando all'istante di inizio dell'evento successivo.
- La event-list o future event list: la simulazione mantiene almeno una lista di eventi simulati, detta anche pending list o pending event set poiché gli eventi al suo interno sono pendenti, ossia sono stati inseriti nella lista a seguito di eventi verificatosi in precedenza, ma non sono ancora stati simulati. Un evento è caratterizzato da un intervallo di tempo in cui esso si verifica e da un tipo che indica il codice che verrà utilizzato per simulare l'evento. Spesso il codice del tipo associato ad un evento contiene dei parametri, in tal caso vengono inseriti nella descrizione dell'evento; Mentre gli estremi dell'intervallo di tempo rappresentano l'istante di inizio e di fine dell'evento. Nelle simulazioni con engine basati su single-thread, in caso di eventi istantanei si ha solamente un evento corrente, invece, se l'engine supporta il multi-threading, possiamo avere più di un evento corrente, ma questo comporta anche la necessità di avere dei meccanismi di sincronizzazione tra i vari eventi per evitare di avere risultati indesiderati. Questo insieme di eventi pendenti viene solitamente creato utilizzando una coda a priorità, ordinata per tempo iniziale di evento. In questo modo, indipendentemente dal tempo in cui gli eventi vengono inseriti nella coda, essi verranno simulati e rimossi in ordine cronologico. Vengono utilizzati algoritmi general-purpose per gestire la coda in modo efficiente, tra i più noti ci sono l'algoritmo splay tree e l'algoritmo skip list. Lo scheduling degli eventi è dinamico e avviene nel corso della simulazione: nell'esempio precedente, l'evento cliente-arriva si verifica al tempo t e, se la coda dei

clienti è vuota e il commesso non è occupato, implica la creazione di un evento cliente-esce al tempo t+s dove s è il tempo necessario al commesso per servire il cliente.

- Il generatore di numeri casuali: la simulazione ha bisogno di variabili casuali di vario tipo, a seconda del sistema studiato, per essere il più precisa possibile; ma allo stesso tempo, può essere utile far girare la simulazione più di una volta con le stesse variabili per studiare un particolare comportamento del sistema. Questo è possibile grazie all'utilizzo di generatori di numeri pseudocasuali, che nelle simulazioni vengono quindi preferiti ai generatori di numeri casuali veri e propri. Tuttavia, uno dei problemi dovuti all'utilizzo di generatori pseudocasuali nei sistemi a eventi discreti consiste nella non conoscenza a priori della distribuzione dei tempi degli eventi e quindi degli stati stazionari del sistema. Di conseguenza, l'insieme iniziale di eventi posti nella event list non avrà tempi di arrivo rappresentativi del regime stazionario della distribuzione. Questo problema viene risolto sfruttando il bootstrapping del sistema di simulazione: non ci si preoccupa di assegnare valori realistici ai tempi iniziali degli eventi pendenti, perché questi produrranno altri eventi e con il tempo, la distribuzione dei tempi degli eventi prodotti, approssimerà il suo regime stazionario. Alla fine, nella raccolta di statistiche sul sistema simulato, è importante ai fini della correttezza togliere gli eventi che sono avvenuti prima di aver raggiunto il regime stazionario, oppure di far durare la simulazione abbastanza a lungo da rendere il comportamento al bootstrapping trascurabile rispetto al comportamento in regime stazionario.
- Le statistiche: la simulazione tiene solitamente traccia delle statistiche del sistema simulato, quantificandone gli aspetti di interesse. Ad esempio, nel caso precedente, potrebbe essere interessante tracciare il tempo di attesa medio dei clienti. In generale, nei modelli di simulazione, la metrica utilizzata non viene derivata dalla probabilità di distribuzione, ma dalla media ottenuta replicando la simulazione, utilizzando degli intervalli di confidenza per accertare la qualità dell'output.
- L'ending condition: la simulazione ha bisogno di una condizione di terminazione, perchè gli eventi sono gestiti automaticamente dalla simulazione, quindi essa potrebbe potenzialmente girare all'infinito. La scelta della condizione viene fatta da chi progetta il simulatore, che quindi decide quando la simulazione termina; la scelta del valore viene invece lasciata all'utilizzatore. Tipicamente, le condizioni utilizzate sono quando il current time raggiunge il tempo t scelto, quando sono stati processati n eventi, quando una certa statistica raggiunge il valore X.

Quindi un sistema per essere simulabile a eventi discreti deve soddisfare i seguenti requisiti:

- Deve mantenere una event list.
- Deve poter permettere la creazione di un event record, ossia una struttura dati che contiene le informazioni utili sull'evento; è importante che ci sia almeno l'event time. Deve inoltre permettere l'inserimento e la cancellazione dalla lista.
- Deve mantenere il clock della simulazione, ossia il current time.
- Se si desidera una simulazione stocastica, deve fornire strumenti per generare numeri casuali dalle distribuzioni di probabilità comuni.

1.3 La logica

Il ciclo principale di una simulazione a eventi discreti si compone generalmente di 3 fasi:

- La fase di start: la condizione di terminazione viene inizializzata a false, vengono inizializzate le variabili di stato, il clock viene settato al tempo di inizio della simulazione (solitamente a zero), vengono messi nella event list gli eventi iniziali.
- La fase di Loop: solitamente si utilizza il costrutto Do...While oppure solamente il While; il ciclo continua fino a che non si verifica la condizione di terminazione. Ad ogni passo: il clock viene settato pari al tempo di inizio del primo evento nella coda, l'evento viene simulato e rimosso dalla coda, infine vengono aggiornate le statistiche.
- La fase di end: la simulazione termina generando un report con le statistiche del sistema simulato.

Più in generale, sono possibili 3 approcci: activity scanning, event scheduling, process interaction.

Nell'activity scanning il building block è l'attività o operazione, i segmenti di codice del model program sono formati da una sequenza di attività che aspettano di essere eseguite e, per fare ciò, dev'essere soddisfatta la condizione posta sulla sequenza di attività. Il simulator si sposta da un evento all'altro ed esegue le sequenze di attività la cui condizione è soddisfatta.

Nell'event scheduling il building block è l'evento, i segmenti di codice sono formati da routines legate agli eventi che aspettano di essere eseguiti. Ogni routine è associata ad un tipo di evento ed esegue le operazioni specifiche di quel tipo. Il simulator si sposta da un evento all'altro eseguendo le routines associate.

Nel process interaction il building block è il processo, il sistema è composto da un insieme di processi che interagiscono tra di loro e i segmenti di codice di ogni processo includono le operazioni che il processo esegue durante il suo ciclo di vita. Nel sistema, la sequenza di eventi è composta dalla fusione di tutte le sequenze di eventi dei singoli processi e la event list è composta da una sequenza di event nodes, ossia una struttura che contiene l'event time e il processo a cui l'evento appartiene. Il simulator si fa carico dei seguenti task: l'inserimento dei processi in un certo punto temporale nella event list, la rimozione dei processi dalla event list, l'attivazione del processo corrispondente al prossimo event node della event list, il rescheduling dei processi in lista. Solitamente un processo può trovarsi in 4 stati:

- Active: il processo è attualmente in esecuzione, in un sistema possiamo avere un solo processo active.
- Ready: il processo si trova nella event list, in attesa di essere eseguito in un certo istante.
- Idle o blocked: il processo non si trova nella event list, ma può essere riattivato da un'altra entità, ad esempio un processo che attende una risorsa passiva.
- Terminated: il processo ha terminato la sua sequenza di operazioni, non si trova nella event list e non può essere riattivato.

Ogni processo è composto da un segmento di codice, ogni segmento viene implementato utilizzando una coroutine, ossia una routine sequenziale che ad un certo punto si sospende, per essere ripresa dopo, in un momento preciso. Anche se molte coroutine possono cooperare, ne esiste una sola attiva. Se i thread sono supportati, ossia i processi leggeri, essi vengono utilizzati al posto delle coroutines perché, a differenza di queste, possono davvero eseguire le operazioni simultaneamente, a patto che sia disponibile hardware di tipo parallelo. L'algoritmo generico di questo tipo di approccio è il seguente:

```
while (la event list non è vuota && la simulazione non è terminata){
    get(prossimo event node nella event list);
    currentTime = eventTime;
    attivare la coroutine del processo corrente dell'event node;
    eseguire le attività dell'event node;
    aggiornare l'indicatore di riattivazione indicando la prossima fase del processo e
        inserire l'event node corrispondente nella event list;
    disattivare la coroutine corrente;
```

}

I punti di riattivazione delle coroutines sono i punti di partenza delle diverse attività legate agli eventi, ogni sequenza di attività è una fase ed ogni fase viene eseguita in tempi diversi (in caso di thread vengono eseguite simultaneamente).

Possiamo quindi creare un modello object oriented in cui: le entità attive vengono rappresentate da oggetti, i tipi di processo vengono rappresentati da classi che implementano i comportamenti dei processi, che deriveranno da una classe base process nel package della libreria del simulator. Pertanto l'istanza di un processo è rappresentata dall'istanza, e quindi un oggetto, di una classe, gli attributi del processo come gli attributi di tale istanza, il suo comportamento è definito dai metodi di tale istanza. Le risorse, che sono entità passive, sono rappresentate da classi, che non mostrano il comportamento del processo, infatti queste classi non deriveranno dalla classe process di base, ma da un'altra classe di base resource, o da un'interfaccia. Infine il programma della simulazione dovrà contenere classi utili al fine di collezionare statistiche o report.

1.4 Possibili applicazioni

La simulazione a eventi discreti può essere applicata a tutti i sistemi che possono essere modellati rispettando le caratteristiche della simulazione a tempo discreto citate nei paragrafi precedenti. Questo approccio è particolarmente adatto nel diagnosticare i problemi relativi a processi complessi: è importante infatti capire dove si trovano i bottlenecks, ossia i colli di bottiglia, del sistema; queste zone sono critiche in quanto diminuiscono notevolmente le prestazioni del sistema e solitamente sono associate alle componenti più lente. L'unico modo per migliorare significativamente le prestazioni di un processo, è migliorarlo proprio in queste zone critiche che, purtroppo, in molti processi vengono offuscate dall'eccesso di inventario, dalla sovrapproduzione, dalla diversità dei processi, dalla diversa sequenzialità e stratificazione.

Pertanto, modellando questi processi in modo accurato ed inserendoli all'interno di un simulatore, è possibile ottenere una vista più dettagliata dell'intero sistema, trovando i colli di bottiglia ed utilizzando indicatori di performance per analizzare il sistema e migliorarne le performance. Indicatori tipici possono essere il tempo effettivo di lavoro, numero di consegne puntuali, quantità di scarti, flussi di cassa e così via.

Un esempio di applicazione riguarda l'uso di una sala operatoria di un ospedale: la sala viene condivisa da diverse discipline medico-chirurgiche e, studiando meglio le procedure associate a queste, si può aumentare il throughput di pazienti, ossia il numero di pazienti operati in un certo periodo di tempo. Ad esempio, se un'operazione al cuore dura in media 4 ore, aumentare le ore in

cui la sala è disponibile da 8 al giorno a 9 non aumenterebbe il throughput, poiché si avrebbero al massimo due interventi come nel caso delle 8 ore. D'altra parte, se una procedura, invece, richiede 20 minuti, fornire un'ora aggiuntiva potrebbe non comportare alcun aumento del throughput se il tempo trascorso nella stanza è trascurabile rispetto alla durata dell'intervento, perché l'intervento durerebbe meno e quindi in 20 minuti si opererebbe un solo paziente. Però tenendo conto di entrambe le procedure, con un'ora aggiuntiva possiamo avere 3 interventi brevi da 20 minuti e 2 interventi al cuore da 4 ore ciascuno, ordinando gli interventi per durata minore e inserendoli nella event-list, producendo un aumento del throughput di pazienti giornalieri.

Un altro esempio di applicazione lo troviamo nei laboratori in cui vengono testate le performance del sistema prima e dopo un'ipotesi di miglioramento di una sua componente: l'impatto sull'intero sistema non sempre è prevedibile, grazie alla simulazione a eventi discreti è possibile vedere i cambiamenti che il miglioramento di una singola componente ha su tutto il sistema.

Altre applicazioni le troviamo in ambito economico, ad esempio la simulazione viene spesso utilizzata per valutare i possibili investimenti di capitale, aiutando gli investitori a valutare le potenziali alternative e a prendere delle decisioni più razionali.

Infine, possiamo trovare un'applicazione anche nelle reti, in cui la simulazione a eventi discreti viene utilizzata per simulare nuovi protocolli su diversi scenari di traffico prima che questi vengano rilasciati.

1.5 Il confronto con la simulazione reale e con altre tecniche

Una simulazione a eventi discreti è un modello virtuale che imita le operazioni di un sistema realmente esistente, come ad esempio le operazioni giornaliere di una banca, il processo di una linea di assemblaggio di una fabbrica, l'assegnazione di personale in un ospedale o in un call center. Per questo motivo, deve tenere in considerazione tutti le risorse e i vincoli legati al sistema, e le loro interazioni nel tempo. Deve, inoltre, combaciare il più possibile con la realtà, e quindi considerare che gli eventi possono avere tempo di completamento variabile e questo viene fatto introducendo i generatori pseudocasuali di cui abbiamo parlato prima; in questo modo la simulazione è molto più vicina alla situazione reale e quindi attraverso di essa si può prevedere il comportamento del sistema a fronte di cambiamenti nei dati di input o nella sua struttura, proprio come se la simulazione fosse reale. Pertanto, con questo tipo di simulazione, è possibile testare le proprie idee molto più rapidamente e con un costo di molto inferiore rispetto alla simulazione reale; questo

risparmio di tempo e denaro può dare spazio ad altre idee e quindi ad un probabile miglioramento nella gestione del sistema.

In genere, ogni sistema reale che è esprimibile attraverso un flusso di processi con eventi, può essere simulato. I processi da cui si può trarre maggior beneficio dalla simulazione, sono quelli con più cambiamenti nel tempo e con un alto livello di casualità. Un buon esempio riguarda una stazione di rifornimento per veicoli: non si può prevedere esattamente quando arriverà il prossimo cliente, e nemmeno il tipo e la quantità di carburante che richiederà.

Di conseguenza la simulazione a tempi discreti, ove applicabile, va preferita alla simulazione reale, poiché si ottiene un guadagno in termini di costo, di tempo e di ripetibilità della simulazione. Basti pensare al costo associato ad ogni simulazione reale, non solo per quanto riguarda la spesa nell'assumere nuovi dipendenti o acquistare nuove componenti, ma anche alle conseguenze che queste decisioni possono avere sul sistema reale, che possono portare a risultati positivi, ma anche negativi. Questo è invece evitabile nella simulazione virtuale in cui il sistema non ha connessioni con il sistema reale, e quindi il test non provoca conseguenze economiche. Inoltre, nella realtà è più difficile simulare due volte con le stesse condizioni lo stesso sistema, mentre nella simulazione virtuale si può testare un sistema con le stesse condizioni modificando solamente l'input, e questo da certezze maggiori sul fatto che un'idea sia migliore di un'altra. Infine come detto prima il tempo necessario ad effettuare una simulazione virtuale è decisamente minore rispetto a quello di una simulazione reale, soprattutto se il sistema simulato ha un periodo temporale molto lungo, ad esempio se vogliamo analizzare il throughput di clienti in un mese, nella simulazione reale deve passare almeno 1 mese, in quella virtuale bastano pochi secondi.

Per quanto riguarda le altre tecniche, esistono tools matematici che riescono a gestire dei modelli in regime stazionario, ma non riescono a gestire gli eventi casuali, ad esempio, la rottura di una delle macchine in una catena di montaggio; inoltre, se il sistema è complesso, spesso questi tools falliscono e la simulazione è l'unica alternativa valida.

Un'altra tecnica consiste nello sfruttare le distribuzioni: ad un evento viene associato il tempo medio di esecuzione, ma nella realtà il tempo può variare; ad esempio, il tempo necessario per servire un cliente dipende dalla quantità di oggetti che egli vuole acquistare che può variare molto dalla media; invece la simulazione usando generatori pseudocasuali si avvicina molto di più alla situazione reale.

1.6 Software disponibile

Ci sono molti prodotti commerciali che forniscono un ambiente di simulazione a eventi discreti, sia di tipo commerciale che open source, implementati utilizzando sia linguaggi di programmazione comuni, sia linguaggi specifici. Di seguito riporto la lista dei software commerciali più famosi:

Software	Descrizione
AnyLogic	Un tool general purpose con metodi di applicazione multipli.
Arena (software)	Un programma per simulazioni a eventi discreti che è applicabile anche a processi continui.
Care pathway simulator	Un programma per simulazioni a eventi discreti progettato specificatamente per industrie di servizi, come ad esempio l'assistenza sanitaria.
Enterprise Dynamics	Una piattaforma con software per simulazioni.
ExtendSim	Un package con software per simulazioni general-purpose.
FlexSim	Un software per simulazioni a eventi discreti con interfaccia drag-and-drop per costruire il modello 3D della simulazione.
GoldSim	Combina la dinamica del sistema simulato con gli aspetti della simulazione a eventi discreti, è stato inserito nella struttura di Monte Carlo.
GPSS	Un linguaggio per simulazioni a eventi discreti, sono disponibili diverse implementazioni a seconda del vendor.
MS4 Modeling Environment	Un ambiente software general purpose basato sulla metodologia DEVS per eventi discreti e modelli ibridi.
Plant Simulation	Un Software che consente la simulazione e ottimizzazione di sistemi di produzione e processi.
ProModel	Un tool per simulazioni a eventi discreti applicabile anche a processi continui.
Simcad Pro	Un software per simulazioni dinamiche, sia continue che a eventi discreti discrete. Fornisce un'interfaccia grafica e non necessita di un ambiente per scrivere il codice.
SimEvents	Aggiunge all'ambiente di simulazione Simulink di MATLAB un simulatore a eventi discreti.
Simio	Un software per simulazioni a eventi discreti object oriented agent-based
SIMUL8	Un softare per simulazioni Object-based.
VisualSim	Un sistema model-based per l'esplorazione dell'architettura di apparecchiature elettroniche, software embedded e semiconduttori, basato sul timing, sul consumo di energia elettrica e sulla

Software	Descrizione
	funzionalità.
WITNESS	Simulatore a eventi discreti con VR disponibile sia su desktop che su cloud.

Tra i prodotti open source, i più noti sono e i rispettivi linguaggi di programmazione utilizzati sono:

Software	Linguaggio	Descrizione
CPN Tools	BETA	Un tool per analizzare l'accodamento e la logistica di un modello, in ogni tipo di applicazione.
DESMO-J	Java	Un framework di simulazione a eventi discreti in Java, supporta anche eventi e processi ibridi e fornisce un'animazione in 2D e in 3D.
Facsimile	Scala	Una libreria per simulazioni e emulazioni a eventi discreti.
PowerDEVS	C++	Un tool integrato per la costruzione di sistemi ibridi e simulazione, basato sul formalismo DEVS.
Ptolemy II	Java	Un software con framework che supporta la sperimentazione con design actor-oriented.
SIM.JS	JavaScript	Una libreria general purpose per simulazioni a eventi discreti scritta interamente in Javascript. Viene lanciata su un browser e supporta anche una GUI per la costruzione del modello.
SimPy	Python	Un framework per simulazioni di processi e eventi discreti basato su Python standard.
Simula	Simula	Un linguaggio di programmazione progettato esclusivamente per le simulazioni.
SystemC	C++	Un set di classi C++ e macro che forniscono un kernel per simulazioni event-driven.

Tra i prodotti citati, quelli che rispecchiano l'attuale stato dell'arte sono Arena e Flexsim, di cui parleremo nel dettaglio nel prossimo capitolo, ma, per la sua importanza storica e poiché l'ambiente che abbiamo sviluppato utilizza il linguaggio Java, mi sembra opportuno citare parlare anche di JaamSim.

1.6.1 JaamSim

JaamSim fu il primo software per simulazioni open source ad offrire un'interfaccia drag-and-drop in grado di competere con i prodotti di tipo commerciale. E' un ambiente di simulazione a eventi discreti in java, è stato sviluppato nel 2002 e fornisce un'interfaccia drag-and-drop, una grafica interattiva in 3D, input e output processing, tools e editors per la costruzione del modello da simulare. La sua caratteristica principale che lo differenzia dagli altri prodotti commerciali è la possibilità, da parte dell'utente, di sviluppare un set di oggetti di alto livello per la propria applicazione. Questi oggetti avranno automaticamente una grafica 3D, saranno disponibili nell'interfaccia drag-and-drop e i loro input possono essere modificati attraverso l'input editor.

Quindi gli utenti possono focalizzarsi sulla logica dei propri oggetti, senza dover programmare un'interfaccia utente e il processing dell'input e dell'output. Gli oggetti vengono definiti usando Java standard, in tool di sviluppo standard come Eclipse; non servono linguaggi per simulazioni specifici, non servono diagrammi di flusso o linguaggi di scripting come quelli usati nei prodotti commerciali. La logica del modello può essere codificata all'interno di un evento o di un processo usando classi e metodi forniti da JaamSim che raccolgono le principali funzioni per la simulazione: oggetti grafici, overlay di testo, il clock, frecce, grafi, distribuzioni di probabilità, generatori, code, risorse, somme pesate, polinomiali e così via. Per quanto riguarda le prestazioni, JaamSim è veloce è scalabile rispetto ai prodotti commerciali sopra citati che fanno uso di un'interfaccia drag-and-drop.

Capitolo 2

Stato dell'arte: FlexSim e Arena

2.1 Introduzione

Nel capitolo precedente, abbiamo descritto le caratteristiche e le componenti necessarie che un sistema deve avere, per poter essere simulato in un ambiente di simulazione a eventi discreti e abbiamo elencato una serie di prodotti che forniscono del software utile per effettuare tali simulazioni, sia di tipo open source che ti tipo commerciale. In questo capitolo, invece, vedremo a che punto si trova lo stato dell'arte, andando ad esaminare due prodotti software all'avanguardia: FlexSim e Arena.

2.2 FlexSim

FlexSim è un software per simulazioni 3D che può creare, visualizzare e simulare svariati tipi di sistemi industriali, tra cui la gestione del materiale, la gestione del magazzino, la logistica, l'assistenza sanitaria, sistemi d'estrazione e di produzione. Andiamo ad esaminare questi sistemi nel dettaglio.
Con il termine produzione, intendiamo un sistema di produzione reale simulato e rappresentato secondo i principi della simulazione a eventi discreti. Flexsim fornisce alle industrie uno strumento in grado di analizzare e testare i processi di produzione in un ambiente virtuale, riducendo i requisiti di tempo e denaro associati ad una simulazione reale. L'inventario, la fase di assemblaggio, di trasporto dei materiali e della produzione può essere rappresentata all'interno del modello simulato, ottenendo dell'informazione utile per aumentare il guadagno ad un costo molto basso.
Nel settore industriale, infatti, la necessità di maggior efficienza, in queste fasi della produzione, è molto richiesta, tenendo presente che il costo del lavoro aumenta di anno in anno. Le aziende di successo devono assicurarsi che il costo associato al tempo, ai macchinari e agli investimenti venga considerato e ottimizzato. Sostanzialmente, la simulazione del sistema produttivo è a costo zero e offre un modo senza rischi per testare ogni possibile modifica da apportare al sistema, da una semplice sostituzione ad un re-design dell'intero sistema, cercando di soddisfare gli obiettivi e minimizzare i costi. Inoltre, questo tipo di simulazione offre uno strumento rapido ed efficiente per

configurare i parametri e ri-simulare lo stesso sistema, risparmiando così del tempo prezioso e velocizzando i risultati.
Usando FlexSim, quindi, possiamo trovare risposta a domande importanti sul sistema produttivo; ad esempio possiamo verificare se l'aggiunta di un macchinario nell'impianto incrementa il throughput o se essa crea un bottleneck non previsto; oppure se l'aggiunta di un nuovo processo continuerà a soddisfare gli obiettivi della produzione; oppure possiamo semplicemente analizzare il sistema, identificare i difetti e minimizzarne le cause.
Sempre nel settore industriale, FlexSim viene utilizzato anche per la gestione e trasporto del materiale utilizzato per la creazione dei prodotti, infatti è in grado di modellare sistemi di qualunque dimensione e con un livello di dettaglio molto elevato, aiutando anche qui, ad analizzare i sistemi e a migliorare l'efficienza del trasporto dei materiali nella catena.
Un altro settore in cui viene applicato FlexSim è quello del trasporto del gas, della corrente elettrica o del petrolio; attraverso un modulo chiamato FloWorks. Questo modulo, sviluppato da TALUMIS, partner di FlexSim, rende possibile integrare velocemente e in maniera accurata processi continui in FlexSim. Esso sfrutta un approccio cutting-edge per escludere i calcoli non necessari e può simulare anche le reti più complesse in pochi secondi, calcolando il flusso ottimale della rete. Gli oggetti in FloWorks sono in grado di rappresentare la scissione di un flusso in più flussi e l'unione di più flussi in un unico flusso, impostando delle priorità o associando valori medi ai flussi; questo permette di rappresentare una rete ramificata, identica a quella reale. I flussi possono anche essere spostati utilizzando oggetti Tubo e oggetti Trasportatore, possono essere depositati in oggetti Serbatoio e modificati in caso di guasti imprevisti o in caso avvenga una manutenzione programmata. Tutti gli oggetti hanno al proprio interno un meccanismo di innesco che si attiva al raggiungimento di determinati livelli di serbatoio, di volumi prodotti o di volumi ricevuti; questo consente al simulatore di modellare accuratamente il sistema reale in modo rapido ed affidabile.
Per quanto riguarda l'industria mineraria, Flexim Software Products, Inc. ha collaborato con la società di consulenza mineraria RungePincockMinarco (RPM) per realizzare un prodotto software per l'industria mineraria. Questo prodotto si chiama HAULSIM ed è un pacchetto per simulazioni di trasporti ed è attualmente lo strumento più potente disponibile per creare, analizzare, osservare e ottimizzare una rete di trasporto, garantendo affidabilità e precisione dei piani minerari. Veloce da configurare, facile da utilizzare ed è la prima soluzione di simulazione di trasporto specifica del settore minerario.
Nel settore sanitario, invece, è stato sviluppato FlexSim HC (Healthcare) ed è il software di simulazione più potente, completo e facile da usare al mondo, progettato specificamente per le strutture sanitarie di oggi. Esso consente l'analisi di tutti le componenti della struttura e dei risultati dei pazienti per valutare l'impatto che hanno sul sistema sanitario, il tutto in un ambiente virtuale

incentrato sul paziente. Le soluzioni trovate sono accurate e possono aiutare l'organizzazione sanitaria a essere il più efficiente possibile senza il rischio di sperimentazioni nel mondo reale e senza sacrificare risorse. Le immagini 3D complete di FlexSim HC, che utilizzano la più recente grafica OpenGL, consentono di vedere esattamente cosa sta succedendo nel modello durante la sua esecuzione perchè è molto più facile confrontare i risultati vedendo visivamente cosa sta accadendo al sistema sanitario simulato. Le immagini 3D creano anche maggior opportunità di comunicazione e lavoro di squadra. Una simulazione con accurati modelli 3D del personale, delle attrezzature e dei mobili presenti nella struttura sanitaria riesce ad essere più convincente rispetto ad una semplice tabella.
FlexSim è molto potente ed applicabile a moltissimi altri sistemi, è user-friendly, aiuta ad ottimizzare i processi pianificati e correnti, identificare ed abbassare lo spreco di risorse, ridurre il costo e di conseguenza incrementare il guadagno. Secondo alcuni clienti, è il miglior software per simulazioni in circolazione e il software viene sviluppato seguendo i bisogni dei clienti stessi; inoltre, l'interfaccia drag-and-drop e la grafica 3D semplificano l'utilizzo del software e permettono di individuare con facilità i bottlenecks all'interno della sequenza di processi del sistema.
Di seguito vedremo alcuni casi studio realistici in cui FlexSim:

- Ha aiutato un produttore high-tech del mercato globale a sviluppare un nuovo processo di pianificazione strategica per prendere decisioni rapide e affidabili nel momento in cui bisogna scegliere se mantenere dei siti produttivi o come raggruppare i volumi a livello globale.
- Ha creato un modello di simulazione di lavoro di un magazzino automatico da utilizzare come strumento di supporto decisionale. Il risultato è stato uno strumento flessibile per rispondere a domande critiche sulla struttura attuale, ma anche in grado di simulare la struttura a seguito di cambiamenti futuri.
- Ha aiutato un coordinatore di sistemi per la movimentazione di materiali che voleva testare una modifica proposta a un circuito AS / RS con un trasportatore. Quello che hanno trovato è stato un miglioramento significativo del throughput nei processi di immagazzinamento dei loro clienti.

2.2.1 Caso studio: flessibilità produttiva

Nel primo caso studio si vuole trasformare, in modo flessibile, la produttiva di un'azienda per ottenere un vantaggio competitivo; questo perché le aziende più competitive valutano e trasformano

continuamente il modo in cui utilizzano le loro risorse e spesso la trasformazione diventa un'esigenza per massimizzare le risorse in risposta ai cambiamenti circostanziali. In questo caso, è capitato ad un cliente di Quadrillion Partners, egli ha subito la chiusura di un impianto che ha richiesto lo spostamento dei volumi di produzione in altre posizioni, minacciando l'aumento dei tempi di consegna nei confronti dei clienti. La Quadrillion, società di consulenza con sede a Dallas, ha aiutato il proprio cliente sviluppando un nuovo processo di pianificazione strategica per prendere decisioni in modo rapido e affidabile, nel caso in cui vengano consolidati siti di produzione o quando c'è la necessità di un raggruppamento dei prodotti a livello globale. Quadrillion ha quindi collaborato con FlexSim allo sviluppo di un modello di simulazione globale che avrebbe aiutato il proprio cliente in questo processo decisionale critico.

Quindi, l'obiettivo finale, è quello di determinare le opzioni per raggruppare i volumi, per tipologia di prodotto, tra diversi stabilimenti a livello globale, considerando i vincoli legati ai costi, tempi di consegna e al numero di clienti. Questo consentirebbe di capire dove produrre i vari tipi di prodotto, quali costi e quali benefici si avrebbero dall'approvvigionamento di prodotti da diverse regioni e se delle decisioni possono portare a problemi di capacità o alla creazione di bottlenecks.

L'ecosistema produttivo globale è complesso ed è composto da 6 fabbriche che producono e spediscono più di 75.000 articoli e più di 20 tipologie di prodotto, che devono soddisfare più di 600.000 ordini globali all'anno. Le fabbriche utilizzano più di 1.000 parti di apparecchiature, dozzine di linee di produzione a flusso continuo e in alcuni casi addirittura diverse procedure per la produzione di prodotti simili in luoghi diversi. Inoltre, bisogna considerare la fase di consegna ai clienti, riportando nella simulazione ogni aspetto del sistema, in modo da creare uno scenario realistico e ricavare risultati validi anche per il caso reale.

Ciò significava pianificare un singolo modello simulato che collegasse ordini, posizioni dei clienti, posizioni degli impianti, costi e capacità delle strutture; inoltre bisognava considerare tutti gli aspetti dei processi: dalle velocità delle macchine alle esigenze del personale negli impianti, dalle rotte logistiche alle leggi sulle importazioni e sulle esportazioni.

Quadrillion ha quindi raccolto e preparato una grossa quantità di dati operativi sul sistema, e li ha inseriti in 22 fogli di calcolo, aggiornabili anche per simulazioni future sfruttando le funzionalità di importazione di Excel di FlexSim.

La simulazione ha quindi fatto largo uso di Process Flow, strumento di FlexSim per i processi con flussi che consente di definire le caratteristiche del sistema con fasi di processi semplici da utilizzare, facilitando la comprensione grazie anche alla rappresentazione visiva in 3D del sistema. Questo strumento aggiunge anche flessibilità e reattività agli aggiornamenti dei dati quando vengono considerati diversi scenari di simulazione: ad esempio, quando il cliente vuole valutare gli effetti dell'espansione delle apparecchiature, aggiungendo un numero di macchine a un file Excel, la

logica del modello si aggiornerà automaticamente per aggiungere il numero desiderato di macchine; consentendo una rapida valutazione degli scenari per il processo decisionale.
Poichè il tempo di consegna del cliente era un parametro chiave in questo progetto, il modello è stato alimentato con i dati di geo-localizzazione che collegavano ogni sede del cliente con ogni impianto a livello globale per codice postale. I dati di geo-localizzazione sono stati quindi utilizzati per calcolare i tempi medi di consegna in giorni per ciascun prodotto spedito a ciascun cliente a livello globale. Questi dati sono stati un input fondamentale per il modello simulato con FlexSim, infatti, man mano che i volumi venivano raggruppati, il team poteva esaminare l'impatto sui tempi di consegna dei clienti sia a livello di ordini dei clienti sia in un istogramma per tipologia di prodotto. Il modello ha anche mostrato il compromesso sulla rapidità con cui i prodotti possono essere consegnati a un account rispetto a dove viene effettuato l'ordine.
Il modello di simulazione ha esaminato una varietà di scenari tra cui il raggruppamento del volume all'interno delle regioni, le opzioni per consolidare gli impianti e i cambiamenti derivanti dalle nuove leggi sul lavoro con personale variabile nel corso dei giorni. Tutte le metriche critiche erano presenti nell'output del modello: richiesta di ordini da parte di clienti e prodotti, capacità in entrata e in uscita dall'impianto, cambio dei tempi, tempi di ciclo, tempi di consegna dei clienti, e così via. Questo modello ha rilevato anche un bottleneck, ovvero un collo di bottiglia nella produzione, che si sarebbe potuto rilevare solo in un modello di simulazione che considerava la variabilità e le interdipendenze delle operazioni della vita reale. Una volta identificati nel modello, i bottlenecks potrebbero essere risolti con miglioramenti della velocità delle attrezzature, un ri-bilanciamento della domanda tra diversi siti o la sostituzione di parti di apparecchiature.
Diversi risultati immediati del modello sono stati la decisione di aumentare il raggruppamento dei volumi in tutta la regione asiatica, la decisione di ridurre da sei a cinque il numero di stabilimenti e un'iniziativa per espandere le importazioni a basso costo in regioni a più alto costo per determinate tipologie di prodotto.

2.2.2 Caso studio: processo decisionale critico

Nel secondo caso studio si vuole prendere delle decisioni critiche riguardo alla gestione del magazzino: nei grandi centri di distribuzione automatici, organizzare correttamente e serializzare gli ordini è la chiave per assicurarsi che i prodotti escano in tempo; tenendo conto delle numerose interazioni e dipendenze presenti nei magazzini di oggi. La simulazione è una scelta efficace per comprendere e aiutare a bilanciare questi sistemi.
Per Bastian Solutions, la modellazione della simulazione era un componente fondamentale per aiutare la progettazione dei clienti nella distribuzione dei prodotti e implementare un centro di

distribuzione altamente automatizzato. L'obiettivo era quello di creare un modello di simulazione funzionante della struttura, da utilizzare come strumento di supporto decisionale e il suo team di ingegneri ci è riuscito utilizzando il software di simulazione FlexSim. Questo modello è unico perché fornisce un valore immediato e tangibile al cliente, aggiungendo flessibilità per l'uso futuro. Con un modello informatico funzionante della struttura su cui poter sperimentare, i responsabili delle decisioni possono ottenere informazioni rapide e accurate per indagare sui cambiamenti nelle scomposizioni del prodotto, nei volumi e nei mix degli ordini, nei parametri di rilascio degli ordini per il bilanciamento del carico di lavoro, nella logica del sistema di magazzino,nel personale e nei piani di pianificazione e produttività dell'operatore.

FlexSim, che offre ai modellisti la flessibilità necessaria per personalizzare un modello in diversi modi, era il software ideale, e Bastian Solutions lo ha dotato di un'interfaccia intuitiva e di facile utilizzo: il cliente non ha bisogno di alcuna formazione per utilizzare il software: l'interfaccia personalizzata di Bastian consente di modificare facilmente input e parametri del modello riguardanti il personale, il rilascio degli ordini, la capacità, i tassi di prelievo e altro. Il modello può essere modificato, analizzato e quindi modificato di nuovo in pochi minuti.

Bastian Solutions ha superato le difficoltà incontrate durante la modellazione di una struttura così complessa: il modello doveva accettare 40.000 file di dati SAP e associare questi dati agli elementi in FlexSim; questa è una quantità enorme di dati da importare e gestire. Per risolvere questo problema, Bastian ha utilizzato la funzionalità SQL integrata di FlexSim per scrivere una serie di query che rendevano i dati significativi e utilizzabili nel modello. Inoltre, Bastian ha replicato la logica del proprio Warehouse Execution Software (WES), cioè EXACTA, in FlexSim. Il WES è il "cervello" del sistema, che si occupa della regolazione della spedizione e del flusso degli ordini nella struttura. Lo strumento Process Flow di FlexSim è stato utilizzato per replicare la logica WES e Bastian Solutions ha creato un algoritmo che utilizza i blocchi di attività del flusso di processo per valutare la capacità del sistema rispetto agli ordini presenti nella coda e quindi rilasciare gli ordini in un modo che mantenga il sistema equilibrato. Questo algoritmo è abbastanza avanzato da considerare la priorità degli ordini e i vincoli di sistema quando rilascia e instrada i container in tutta la struttura ed è anche semplice leggere e seguire l'algoritmo grazie alla grafica 3D.

Infatti, Bastian Solutions è stata anche in grado di gestire con successo la comunicazione tra la logica creata utilizzando Process Flow e lo spazio 3D, fondamentale per dimensionare l'impianto e i suoi trasportatori e anche per analizzare visivamente la situazione. In particolare, Bastian Solutions ha fatto ampio uso della funzionalità "Wait for Event" di FlexSim, che ha definito come un potente framework per la comunicazione e il trasferimento di dati tra la logica del modello e la sua presentazione 3D.

Per aiutare ad analizzare e valutare il modello, Bastian Solutions ha utilizzato la dashboard incorporata in FlexSim e la sua struttura per simulare i quattro output principali dal proprio WES; al termine di una simulazione, attraverso la dashboard si riusciva a rispondere alle seguenti domande come: Qual è lo stato del sistema in questo momento? Cosa è successo durante l'intera giornata di raccolta? A che velocità lavorano gli operatori e quanto vengono utilizzati? Quanto sono equilibrate le aree di prelievo e le zone di prelievo?
Il modello è stato uno strumento prezioso per il cliente per supportare le decisioni prese nella struttura. Poiché la sua interfaccia personalizzata è stata configurata per cambiare rapidamente dozzine di configurazioni e input, questo modello continuerà a produrre risultati validi per anni. Nelle fasi future del ciclo di vita della struttura, il cliente sarà persino in grado di adattare questo modello, per studiare le modifiche e le espansioni derivanti dalla progettazione del sistema di movimentazione dei materiali.

2.2.3 Caso studio: aumentare il throughput

Nel terzo caso studio si vuole trovare un modo per aumentare il throughput attraverso lo studio di un sistema di archiviazione e recupero automatizzato mediante l'utilizzo di FlexSim. Il model builder di FlexSim ha creato un modello di simulazione del sistema per testare le modifiche proposte alla struttura, con conseguente miglioramento dell'efficienza del magazzino per il cliente finale.
In questo caso Skarnes Inc., un coordinatore di sistemi per la gestione dei materiali con sede a Plymouth, MN, voleva aumentare il rendimento di uno dei processi di magazzino del suo cliente. Il cliente ha utilizzato un sistema di archiviazione e recupero automatico (AS / RS) che si interfacciava con un circuito trasportatore, trasportando i pallet su una stazione di prelevamento e quindi di nuovo sull'AS / RS. Questo sistema era in grado di prelevare 70 pallet all'ora, ma è stato studiato da Skarnes per potenziali miglioramenti.
Dopo aver osservato questa operazione, Skarnes ha ipotizzato che il numero di pallet prelevati all'ora aumenterebbe se la congestione del trasportatore potesse essere ridotta di fronte all'AS / RS. Skarnes ha quindi avuto l'idea di aggiungere un altro trasportatore principale per i pallet in uscita, mettendo il nuovo trasportatore sopra al trasportatore principale, già esistente. Per dimostrare la validità di questa opzione ai propri clienti, Skarnes ha commissionato una simulazione al computer del sistema.
Pertanto, FlexSim ha sviluppato un accurato modello di simulazione 3D per confermare la proposta di Skarnes. Aggiungendo questo trasportatore sopraelevato, il modello di simulazione ha mostrato che il numero di pallet prelevati aumenterebbe da 70 a 100 pallet all'ora, con un incremento del

43% e L'animazione 3D del sistema ha anche confermato visivamente che la congestione del trasportatore è stata notevolmente ridotta, raggiungendo un alto livello di affidabilità per il cliente.

2.3 Arena

Un altro software all'avanguardia, per quanto riguarda le simulazioni a eventi discreti, è Arena Simulation Software, le cui caratteristiche principali sono le seguenti:

- La metodologia di modellazione del diagramma di flusso, include una vasta libreria di blocchi predefiniti per modellare il processo senza la necessità di una programmazione personalizzata.
- Dispone di una gamma completa di opzioni di distribuzione statistica per modellare accuratamente la variabilità del processo.
- Consente di definire percorsi di oggetti e percorsi per la simulazione.
- Consente l'analisi statistica e generazione di report.
- Dispone di metriche di performance e dashboard.
- Possiede funzionalità realistiche di animazione 2D e 3D per visualizzare i risultati oltre ai numeri.

I vantaggi principali forniti da questo software sono i seguenti:

- Migliora la visibilità degli effetti di una modifica sul sistema o su un processo.
- Esplora le opportunità di nuove procedure o metodi senza interrompere il sistema attuale.
- Consente l'analisi diagnostica e la risoluzione dei problemi.
- Permette di ridurre o eliminare i colli di bottiglia (bottlenecks).
- Permette di ridurre i costi operativi.
- Migliora le previsioni finanziarie.
- Consente di valutare meglio i requisiti hardware e software.
- Permette di ridurre i tempi di consegna.
- Permette di gestire meglio l'inventario, il personale, i sistemi di comunicazione e le attrezzature.
- Aumenta la redditività attraverso miglioramenti sulle operazioni.

Queste caratteristiche lo rendono uno dei software più avanzati disponibili oggigiorno. Di seguito andremo ad esaminare alcuni casi studio che trattano di esperienze reali in cui Arena ha aiutato a prendere decisioni critiche e ha consentito un miglioramento del rendimento aziendale.

Il software di simulazione Arena consente alle industrie manifatturiere di aumentare il throughput, identificare i colli di bottiglia del processo, migliorare la logistica e valutare le potenziali modifiche nel processo. Con Arena è possibile modellare e analizzare il flusso del processo, i sistemi di imballaggio, il routing del lavoro, il controllo dell'inventario, il magazzino, la distribuzione e la gestione del personale.

2.3.1 Caso studio: ottimizzazione dei processi produttivi

Il primo caso studio riguarda l'ottimizzazione dei processi produttivi: l'obiettivo è quello di ridisegnare una parte significativa del processo di assemblaggio finale mantenendo gli standard di produzione; in questo caso, il cliente è un importante produttore di elettrodomestici che produce frigoriferi usando una catena lineare.
La soluzione di Arena consisteva nel costruire un modello di simulazione di produzione, altamente dettagliato valutando i flussi dinamici dei prodotti nel sistema, includendo la gestione dei materiali e le operazioni di produzione. L'alto livello di dettaglio era necessario per catturare in modo preciso le risposte del sistema alle operazioni di produzione e l'analisi ha mostrato la quantità di spazio richiesta nel buffer di produzione nei vari scenari e per diverse disposizioni delle apparecchiature. Un'animazione dettagliata ha fornito affidabilità al modello visualizzando ogni flusso mentre attraversava il sistema, lo stato dinamico dei buffer di produzione, e rilevando i colli di bottiglia in esso presenti.
Pertanto, lanciando un programma di produzione attraverso il modello di simulazione, Arena è stata in grado di trovarne uno con la quantità ottimale di spazio nel buffer e le risorse di sistema minime necessarie per raggiungere gli obiettivi di produzione. Grazie al modello, sono stati calcolati vari compromessi in termini di costi, il bilanciamento delle attrezzature e i costi del trasportatore rispetto alla produzione e al volume di produzione.

2.3.2 Caso studio: riduzione dei costi

Il secondo caso riguarda uno dei maggiori produttori mondiali di carta da imballaggio natalizia, responsabile del 30-40% della produzione mondiale, che vuole aumentare la produzione per soddisfare la domanda stagionale. L'obiettivo non è solo quello di aumentare la produttività, ma anche di ridurre l'inventario, ridurre i costi operativi e il tempo di ciclo e migliorare la gestione della capacità del magazzino.
Al fine di identificare i problemi critici nel processo del produttore, i consulenti hanno messo in atto un'iniziativa di analisi e riparazione dei processi aziendali che includeva la simulazione della catena

produttiva. Arena venne selezionato perché è riuscito ad illustrare rapidamente le prestazioni dell'intero processo: la previsione e l'inserimento dei nuovi clienti, la creazione di un database per il prodotto, il completamento della progettazione del pacchetto e la finalizzazione delle informazioni di controllo sulla qualità del prodotto.
Il modello della produzione di Arena è stato utilizzato per valutare oltre 20 scenari di implementazione, identificando almeno cinque che potrebbero migliorare il sistema. Lo scenario definitivo ha dimostrato come ridurre il tempo di ciclo da cinque settimane a tre giorni; abbattere le spese straordinarie, riducendo così le spese operative; e aumentare le prestazioni di spedizione puntuali dal 70% a oltre il 95%. La simulazione ha anche individuato e risolto molti colli di bottiglia interconnessi e ha identificato come le fluttuazioni nel volume o nella tempistica della domanda avrebbero influenzato il sistema. Inoltre, ha identificato l'impatto dell'attività tardiva sul programma di produzione, consentendo all'azienda di ricavare più di un milione di dollari dalle vendite aggiuntive.

2.3.3 Caso studio: progettazione di una nuova linea di produzione

Il terzo caso riguarda una linea di assemblaggio di automobili: le linee di produzione svolgono un ruolo fondamentale e prezioso nel sistema di produzione e sviluppare una linea di produzione efficiente e portarla alla realizzazione è il compito principale di un line designer. Vennero intervistati diversi produttori automobilistici giapponesi e ci si rese conto che le linee di produzione erano tradizionalmente progettate, seguendo metodi convenzionali che per la maggior parte erano basati sull'esperienza ingegneristica e semplici calcoli sull'utilizzo dei lavoratori, sull'utilizzo della macchina e sulla produttività della linea con dati di elaborazione costanti.
L'obiettivo di questa ricerca è di introdurre una nuova prospettiva riguardante la struttura della progettazione di linee di produzione negli impianti automobilistici giapponesi e per raggiungerlo venne utilizzato il software Arena, sfruttando la simulazione per studiare i processi di progettazione attraverso casi studio che riguardano la progettazione di nuove linee di produzione.
Per fare questo, era necessario comprendere le nuove informazioni sul progetto, decidere se creare o acquistare le componenti della linea, progettare e creare la nuova linea di produzione, controllare i risultati effettivi della linea progettata e consegnarli alla fabbrica. Infine bisognava fare indagini di mercato sulla domanda per il prodotto e ri-progettare, modificare e riconsegnare alla fabbrica la linea.
Dopo aver progettato la linea di produzione, le attrezzature e i robot sono stati acquistati e fabbricati; sono stati necessari circa 8 mesi per realizzare la linea di produzione. Dopo aver regolato

le macchine, i robot e le parti per garantire una condizione di buona qualità, i membri del progetto hanno effettuato una prova reale: cinque giorni lavorativi di prove per riconfermare la fattibilità della linea (a causa di limiti di tempo e di budget, non è stato possibile eseguire una prova lunga quanto la prova virtuale effettuata attraverso la simulazione). I risultati erano molto vicini ai risultati della simulazione grazie all'elevata abilità dei lavoratori e il tempo di guasto reale era più piccolo rispetto quello della simulazione. I risultati hanno anche dimostrato che la linea di produzione scelta ha soddisfatto l'obiettivo di progettazione e che quindi questa linea poteva essere consegnata alla fabbrica per le normali attività di produzione.
La simulazione è stata molto utile per studiare la fattibilità di nuove linee. Questo studio attraverso la simulazione ha anche contribuito a ridurre i tempi di progetto nel processo di progettazione (il 10% del tempo totale) e a ridurre i costi nella modifica della linea (5% del costo totale dell'investimento delle attrezzature).

2.3.4 Caso studio: rappresentazione di un sistema reale

L'ultimo caso studio riguarda l'utilizzo di Arena per aumentare la produttività in una fabbrica di pneumatici: l'obiettivo è di trovare un modo per migliorare l'efficienza complessiva e ridurre i costi del design.
Conclusioni sostanziali sono state ottenute dall'analisi: nel trasportatore progettato sono stati identificati due colli di bottiglia in due diversi punti di connessione, i quali avrebbero ridotto del 12% la capacità complessiva del sistema se il progetto fosse stato implementato. Inoltre, tutta la capacità di riserva integrata nel sistema (circa il 15%) sarebbe stata sacrificata a causa dei vincoli del sistema di trasporto. Anche la posizione dei colli di bottiglia si è rivelata critica. Il collo di bottiglia principale del trasportatore era a valle dell'area del processo che, per progettazione, era già destinata a costituire il fattore limitante del sistema che, secondo le statistiche raccolte dai modelli, avrebbe perso circa il 25% della sua capacità a causa del blocco sulla connessione.
Il progetto è stato un successo perché il modello Arena ha replicato, in modo molto dettagliato, come funzionerebbe il sistema reale: i punti di connessione del trasportatore nel modello simulato sono stati creati imitando fedelmente il loro funzionamento nel sistema reale. Questa precisione è stata fondamentale per convincere il cliente che, senza questo livello di realismo, avrebbe probabilmente installato un design inferiore.

Capitolo 3

Standard per simulazioni con BPMN

3.1 Introduzione

Nel capitolo precedente abbiamo descritto le caratteristiche principali di due software allo stato dell'arte: FlexSim e Arena; esaminandone i relativi casi studio. Essi sono applicabili nella maggioranza dei processi industriali, ma né FlexSim né Arena, sono applicabili ai processi business-to-business, infatti, attualmente, non esiste un ambiente di simulazione per processi di business B2B.
Un processo di business deve raggiungere un obiettivo attraverso il coordinamento di unità di lavoro, dette task, e queste possono essere svolte da persone che rivestono un particolare ruolo (human task) oppure dal sistema in automatico (automatic task). La gestione di questo tipo di processi fa parte del Business Process Management (BPM), una disciplina che include la modellazione, l'automazione, l'esecuzione, il controllo, la misurazione e l'ottimizzazione dei flussi d'attività del processo di business; rispettando gli obiettivi e l'organizzazione aziendale.
Ogni processo di business è definito da un Process Model che si basa su una specifica notazione, ed esistono due tipi di processi di business (BP): i processi con una procedura standard(detti anche routine) in cui il coordinamento viene effettuato automaticamente seguendo le regole definite nel process model e i processi specifici che dipendono dal caso preso in questione (detti anche case handler) in cui il coordinamento viene effettuato dai partecipanti coinvolti nel caso. Se il processo è di tipo routine, possiamo utilizzare BPMN per costruire il suo process model.
Business Process Model and Notation (BPMN) è uno standard per la modellazione dei processi di business che fornisce una notazione grafica per specificare i processi di business in un Business Process Diagram (BPD), basato su una tecnica di flowchart molto simile ai diagrammi di attività di Unified Modeling Language (UML). L'obiettivo di BPMN è supportare la gestione dei processi aziendali, sia per utenti tecnici che per utenti aziendali, fornendo una notazione intuitiva, ma in grado di rappresentare una semantica di processo complessa. La standard BPMN fornisce anche una mappatura tra la grafica della notazione e i costrutti sottostanti dei linguaggi di esecuzione.
L'obiettivo principale di BPMN è fornire una notazione standard facilmente comprensibile da tutti i portatori d'interesse aziendale (azionisti, fornitori, investitori e distributori) o più in generale, gli stakeholders. Questi includono anche gli analisti di business che creano e perfezionano i processi,

gli sviluppatori tecnici responsabili della loro implementazione e i business manager che li monitorano e gestiscono. Di conseguenza, BPMN funge da linguaggio comune, colmando il divario di comunicazione che si verifica frequentemente tra la progettazione e l'implementazione dei processi aziendali.

3.2 BPSim: Business Process Simulation

Per quanto riguarda la simulazione di processi di business, non è ancora stato definito uno standard, ma è stato sviluppato un framework chiamato BPSim che ne definisce le specifiche, consentendo di integrare i modelli di processo aziendali acquisiti in BPMN con informazioni per supportare rigorosi metodi di analisi. Definisce quindi la parametrizzazione e l'interscambio dei dati dell'analisi del processo consentendo l'analisi strutturale e capacitiva del process model, supportando sia l'ottimizzazione pre-esecuzione che l'ottimizzazione post-esecuzione. La specifica consiste in una rappresentazione di basso livello, interpretabile dal calcolatore (il meta-modello) e un file di accompagnamento, in formato elettronico, per facilitare la protezione e il trasferimento di questi dati tra diversi tools (il formato di interscambio). Infatti BPSim si focalizza sui dati di input e di output e sui loro formati, piuttosto che sulla loro interpretazione o sul loro uso. Il meta-modello di BPSim viene catturato utilizzando l'UML (Unified Modeling Language) e il formato d'interscambio è definito utilizzando un XML Schema Definition (XSD) ed essi rappresentano il nucleo di questa specifica.
Lo scopo di BPSim è quello di incoraggiare una più ampia adozione della simulazione attraverso un approccio guidato, complementare agli standard, perchè la simulazione di processi è un'ottima tecnica per supportare il designing del processo, riducendo i rischi legati ad eventuali cambiamenti e migliorando l'efficienza dell'organizzazione. Inoltre, vuole favorire l'interscambio tra i dati, per facilitarne la comprensione, aumentarne l'adozione e garantire l'interoperabilità; creando così un mercato aperto, con costi ridotti e senza più vincoli legati ai vendors, permettendo così una scelta più libera, basata sul prodotto che maggiormente soddisfa i requisiti del progetto.
BPSim ha un valore potenziale di business elevato, in quanto:

- supporta la validazione e il designing dei processi.
- Riduce i rischi legati a modifiche nei processi.
- Può predire le prestazioni dei processi di business.
- Fornisce supporto alle decisioni.
- Consente l'allocazione e la gestione delle risorse.

L’approccio BPSim è applicabile sia a process model basati su BPMN sia a PM basati su XPDL e consiste nella parametrizzazione di questi modelli sfruttando prospettive differenti, in modo da poter analizzare, simulare e ottimizzare il processo. Le prospettive disponibili sono definite in termini di costo, tempo, risorse, priorità, proprietà e controllo del processo; poiché le organizzazioni devono affrontare crescenti livelli di pressione per fornire operazioni più efficienti ed efficaci, infatti la simulazione e l'analisi dei processi aziendali vengono riconosciute come parte integrante dell'ottimizzazione delle prestazioni. I processi aziendali inadeguati o mal progettati portano a risultati non attesi dal cliente e ad un comportamento organizzativo indesiderato che può a sua volta portare ad una perdita nelle entrate. Questo è il motivo per cui è importante analizzare a fondo i processi aziendali in un ambiente isolato e sicuro, prima di traslarli nel sistema reale. I vantaggi della simulazione e dell'analisi dei processi aziendali nel testare nuovi processi di business includono: nessun disturbo alle operazioni in corso, la velocità di analisi di possibili scenari e un costo relativo inferiore nell’esplorazione e nella sperimentazione del processo di business.
Nell'analisi di questi processi sono disponibili molte diverse possibilità per migliorare il processo: l'analisi strutturale si concentrerà sugli aspetti strutturali (ad esempio sulla Configurazione) del modello che, di solito, consistono in analisi statistiche che spesso usano metodi statici; invece l'analisi capacitiva lo si concentrerà sugli aspetti capacitivi di un modello (ad esempio sui suoi limiti) ed è, solitamente, basato su analisi dinamiche, spesso utilizzando metodi di simulazione discreti.
Per eseguire tutte queste analisi, i modelli di processi aziendali devono essere integrati con i dati necessari per effettuare tali analisi: nella fase di parametrizzazione, vengono utilizzati sia i valori stimati sia i valori raccolti dopo l’esecuzione, per supportare l’ottimizzazione prima (valori stimati) e dopo (valori raccolti) l’esecuzione.

3.3 BPSim: gli elementi

Nell'analisi del processo BPSim i dati vengono utilizzati per fornire informazioni complementari a un BPMN o XPDL process model, queste informazioni vengono poi utilizzate durante l'analisi, la simulazione e l'ottimizzazione del processo. Il process model dei processi di business e gli elementi dei processi sono, invece, risorse esterne; di seguito andremo ad esaminare gli elementi principali.

3.3.1 Scenario

Gli Scenari all'interno dei dati nell'analisi del processo si riferiscono sempre a un singolo modello di processo di business , eventualmente si possono catturare molti processi in un singolo modello BPMN o XPDL. Quindi il process model è fisso, definito separatamente, con possibili variazioni a seconda dello scenario.
Gli scenari possono essere utilizzati per acquisire:

- Le specifiche dei parametri di input per l'analisi, la simulazione e l'ottimizzazione.
- I risultati dell'analisi, della simulazione e dell'ottimizzazione; gli scenari dei risultati hanno un riferimento allo scenario di input associato.
- I dati storici, relativi all'esecuzione nel mondo reale del modello del processo di business.

Uno scenario è composto da una lista di parametri di elementi: ogni parametro si riferisce ad un elemento che si riferisce ad uno specifico elemento del modello del processo di business. Ogni scenario, inoltre, può avere parametri di scenario, infatti, nei dati BPSim, viene fornito supporto sia agli scenari che ai parametri grazie a soluzioni proprietarie. Uno scenario può, quindi, ereditare la struttura da un altro scenario o diventarne la versione overloaded: in questo caso, devono essere specificate le modifiche ai valori dei parametri degli elementi e i nuovi parametri aggiunti con i rispettivi valori; tutto il resto viene ereditato dallo scenario base.
Gli scenari si trovano all'interno della classe BPSimData che è la classe radice dove tutti gli scenari vengono definiti: essa contiene un attributo scenarios che rappresenta una collezione di oggetti Scenario.
La classe scenario, raggruppa tutti i parametri degli elementi (in oggetti ElementParameter) di uno specifico scenario e ha una serie di attributi utili per descrivere un possibile scenario:

Attributo	Descrizione
String : id	Identifica univocamente lo scenario.
String : name	Il nome di questo scenario.
String : description	La descrizione di questo scenario.
DateTime : created	Indica quando è stato creato questo scenario.
DateTime : modified	Indica quando è avvenuta l'ultima modifica.
String : author	Il nome dell'autore di questo scenario.
String : vendor	Il nome dello strumento software utilizzato per creare questo scenario.
String : version	La versione di questo scenario.

Scenario : result	Nel caso in cui questo scenario fosse l'output di un'analisi e lo scenario sorgente dell'analisi è incluso nel modello, questo campo è un riferimento a tale scenario.
Scenario : inherits	Un riferimento allo scenario base dalla quale questo scenario eredita i valori e i parametri; in caso di overloading, vanno specificati solo i valori e i parametri modificati o non inclusi nello scenario base.
ElementParameter : elementParameter	La collezione di parametri di tipo ElementParameter che compongono questo scenario.
ScenarioParameter : scenarioParameter	Parametri specifici di questo scenario
VendorExtension : vendorExtension	L'estensione proprietaria di questo scenario.

In cui ScenarioParameter è una classe che contiene i parametri specifici: lo start time dello scenario; la sua durata; gli intervalli in cui vanno le statistiche non vengono raccolte; il numero di volte in cui questo scenario va ripetuto; il seed che inizializza il generatore pseudocasuale; l'unità di tempo da utilizzare in questo scenario; la valuta utilizzata per rappresentare i costi; un boolean che indica la volontà di avere un output separato e più dettagliato e la possibilità di definire tale output. VendorExtension invece è una classe che serve per estendere un oggetto Scenario ed è utilizzata dai Vendors per aggiungere dei dati non definiti da BPSim, ma utili ai fini dell'analisi, della simulazione o dell'ottimizzazione. La classe rappresenta una sola estensione, definita attraverso un attributo di tipo String che è formata dal nome del vendor insieme a quello dell'estensione, e da un attributo di tipo Object che indica l'oggetto della classe definita dal vendor che implementa l'estensione desiderata.

3.3.2 Parametri

Ogni ElementParameter di uno scenario, fa riferimento ad un elemento specifico nel modello del processo di business. Per consentire la separazione degli ambiti, gli ElementParameter vengono suddivisi in differenti prospettive, ognuna di esse raggruppa una collezione di parametri che rientrano nello stesso ambito. Il valore di questi parametri può essere accompagnato da un oggetto

Calendar che ne specifica la validità, di cui parleremo nel dettaglio in seguito.
La classe ElementParameter definisce concretamente tutte le prospettive dei parametri, essa è composta da:

Attributo	Descrizione
String : id	Identificativo univoco di questo parametro
BusinessProcessModelElement : elementRef	Un riferimento all'ID dell'elemento del modello del processo di business per cui stiamo definendo un parametro.
VendorExtension : vendorExtension	L'estensione proprietaria di questo parametro.

Opzionalmente, un ElementParameter può avere diversi tipo di oggetto, ad esempio un oggetto PropertyParameters che contiene una lista di Property, oggetto composto da una stringa che ne indica il nome e da un oggetto PropertyType che indica il tipo della proprietà. PropertyParameters definsice le proprietà aggiuntive assegnate agli elementi BPMN e, oltre alla lista, contiene anche un attributo queueLength che indica il numero di token in attesa di risorse disponibili, di tipo Parameter.
Un altro tipo di oggetto inseribile è un TimeParameter che raggruppa tutti i parametri relativi al tempo per un elemento di un processo di business, catturando degli intervalli temporali. L'intervallo più importante è l'elapsedTime, che viene suddiviso in due sotto-intervalli: la duration che indica l'intervallo dall'avvio del processo al suo completamento; lagTime che, invece, indica l'intervallo dal completamento del processo precedente all'avvio del processo corrente. Più precisamente, la duration è data da una somma di altri 4 intervalli: setupTime, processingTime, validationTime e reworkTime; lagTime, invece, è dato dalla somma di transferTime, queueTime e waitTime; se questo livello di granularità non è necessario possiamo assumere la duration pari al processingTime e il lagTime pari al waitTime. Tutti i parametri temporali sono oggetti di tipo Parameter e sono contenuti all'interno di un oggetto TimeParameter.
I valori che questi parametri temporali possono assumere possono essere 3: NumericParameter se si tratta di un numero intero, FloatParameter per numeri con la virgola o DurationParameter se vogliamo indicare un intervallo di valori; a tutti i parametri temporali non definiti verrà assegnato il valore di default 0, tranne che per l'elapsedTime, la duration e il lagTime che possono solamente essere letti nella resultRequest, ma non settati.
Un altro oggetto inseribile è un ControlParameter che raggruppa tutti i parametri che rappresentano il control flow (controllo del flusso) di un elemento del processo: l'interTriggerTime che indica l'intervallo di tempo tra 2 occorrenze dello stesso evento, ossia la frequenza temporale dell'evento (di deafult uguale a 0, l'evento non si verifica mai) ; il triggerCount che indica il numero massimo

di occorrenze (di default è infinito, nessun limite); la probability che indica la probabilità che il controllo passi a questo elemento (di default, per gli eventi è uguale a 0), usata in caso ci siano più flussi d'uscita da un elemento e la decisione su quale flusso seguire viene presa in base al valore di questo parametro (di default, se non viene specificato un valore, questo parametro assume un valore pari alla divisione tra 1 e il numero di flussi uscenti); la condition che indica la condizione da soddisfare per poter passare il controllo a questo elemento (di default è settata a false). Tutti i parametri sono oggetti di tipo Parameter e sono contenuti all'interno di un oggetto ControlParameter.
Un altro oggetto inseribile all'interno di un ElementParameter è un ResourceParameter che raggruppa tutti i parametri che definiscono una risorsa all'interno di un elemento del processo di business: l'availability che ne indica la disponibilità (di default è true), può variare a seconda della lettura di un Calendar (risorsa disponibile solo in alcuni intervalli temporali); la quantity che indica il numero di risorse (di default viene settato a 1); la selection che indica i criteri per la scelta della risorsa, è un override dell'elemento resourceRole del BPMN che assegna un comportamento (ruolo) alla risorsa; il role che indica una lista di ruoli della risorsa (può averne più di uno, di default è null). Tutti i parametri sono oggetti di tipo Parameter e sono contenuti all'interno di un oggetto ResourceParameter, tranne role che è implementato da una lista di Parameter.
Un altro tipo di oggetto che possiamo avere all'interno di un ElementParameter è CostParameter che raggruppa tutti i parametri legati al costo dell'elemento del processo di business: il fixedCost che indica il costo fisso da pagare dopo ogni occorrenza (di default uguale a 0) e lo unitCost che indica il costo variabile, ossia quello da pagare per unità di tempo (di default è uguale a 0). Entrambi sono di tipo Parameter.
Infine all'interno di un ElementParameter possiamo avere un oggetto priorityParameters che raggruppa tutti i parametri che definiscono la priorità dei parametri di un elemento del processo di business: l'interruptible che indica se l'esecuzione di questo elemento è interrompibile (di default è uguale a false) e la priority che determina la priorità di un elemento di un processo di business che viene utilizzata per l'ordinamento in caso di scarsità di risorse (di default è uguale a 0). Entrambi sono di tipo Parameter.

3.3.3 Tipi di parametri

Un Parametro deve avere un valore di default; questo valore può variare a seconda dell'intervallo temporale, poiché è possibile l'assegnazione di valori di aggiuntivi per ogni parametro. Ogni valore aggiunto, deve specificare una fascia temporale di applicabilità, attraverso la definizione di un

Calendar; inoltre, ogni valore aggiunto, deve appartenere ad un tipo specifico.
La classe Parameter è composta da: un attributo di tipo ParameterValue che racchiude le informazioni riguardanti il tipo di parametro; un attributo di tipo ResultType, usato solo per l'input, indica il tipo di risultato che ci aspettiamo di avere nello scenario risultante. ResultType è un Enum che può assumere i seguenti valori in base a ciò che si vuole ottenre nel risultato: min ci fornisce il valore minimo; max il valore massimo; mean il valore medio; count il numero di occorrenze; sum la somma di tutti i risultati. Invece, La Classe ParameterValue possiede 4 attributi: un Calendar che specifica la validità di questo valore (se non specificato, viene assegnato il valore di default); una String che indica l'identificativo univoco del processo da cui questo parametro è stato ricavato; un ResultType che indica il tipo di risultato rappresentato da questo parametro; un DateTime che fornisce il tempo di simulazioni in cui questo risultato è stato prodotto.
Il valore del parametro può essere un ConstantParameter, un EnumParameter, un DistributionParameter o un ExpressionParameter. I ConstantParameter sono oggetti che manterranno lo stesso valore durante tutta la simulazione, possiamo avere un oggetto StringParameter (contiene una String), NumericParameter (contiene un Long + TimeUnit), FloatingParameter (contiene un Float + TimeUnit), BooleanParameter (contiene un boolean), DurationParameter (contiene una Duration) o DateTimeParameter (contiene un DateTime). Un DistributionParameter invece, è un oggetto che cambierà il suo valore nel tempo basandosi su una certa distribuzione: sono disponibili oggetti wrapper che racchiudono le distribuzioni più comuni.

3.3.4 Calendar

Gli oggetti Calendar vengono definiti insieme agli oggetti Scenario e i ParameterValue ne contengono un riferimento. I Calendar vengono creati seguendo il formato iCalendar dell'RFC 5545 e forniscono l'intervallo di tempo per il quale il valore del parametro dev'essere utilizzato. Il formato della Calendar è il seguente:

Attributo	Descrizione
String : id	L'identificativo univoco del Calendar.
String : name	Il nome descrittivo del Calendar.
Object : calendar	Oggetto ottenuto dalla serializzazione XML degli eventi in formato iCalendar con le informazioni descrittive del Calendar.

3.4 Applicabilità

I parametri BPSim non sono applicabili in tutti gli elementi BPMN di un process model. I parametri temporali, ad esempio, sono applicabili solamente agli elementi di tipo Activity, in tutti gli altri elementi (Event, Gateway, Data, Connection Objects, Artifact, Swimlane, Attribute) non è possibile utilizzare questi parametri perché non avrebbero senso nel loro contesto; ad esempio prendendo un elemento Event del BPMN, notiamo che i TimeParameters rappresentano degli intervalli e gli eventi invece vengono mappati su istanti temporali, ossia dei punti.
Anche i ControlParameters non sono sempre applicabili: si possono trovare negli Event, Nelle Activity e nei Gateway; con parecchie limitazioni derivanti dal tipo di parametro di controllo; ad esempio, la condition e la probability non possono essere applicate ad un task, ma l'interTriggerTimer e il triggerCount sono applicabili.
I ResourceParameters possono essere applicati solamente a elementi BPMN di tipo risorse, contenuti in elementi BPMN Attribute, mentre i CostParameters sono applicabili alle risorse BPMN e alle Activity.
Invece, i PropertyParameters possono essere applicati a Event, Activity e Connection Objects di tipo Sequence Flow e Message Flow; mentre i PriorityParameters possono essere applicati solo alle Activity.
Per quanto riguarda gli oggetti ResultRequest, abbiamo detto che abbiamo 5 tipi di risultato possibili: MIN, MAX , MEAN, COUNT e SUM; per ottenere questi risultati bisogna andare a leggere i valori dei parametri e poter effettuare alcune operazione matematiche con essi. Ne deriva che non possiamo ricavare questi tipi di risultato da tutti i parametri. Se consideriamo un TimeParameter, possiamo usare tutti e 5 i tipi, poiché i parametri temporali sono in forma numerica; ma se consideriamo un ControlParameter non avrebbe alcun senso cercare di ottenere uno di questi 5 tipi da una condition o da una probability, mentre può tornare utile ricavare il min e il max di un interTriggerTimerù oppure il count di un triggerCount.
Per quanto riguarda i ResourceParameters possiamo ottenere solamente min e max dalla selection, tutti gli altri tipi non sono ottenibili; mentre nei CostParameters possiamo ottenre solamente la somma sum.
Infine dai PropertyParameters possiamo ottenere solamente min, max e mean dalla queueLenght; mentre dai PriorityParameters non possiamo ottenere nessuno dei 5 tipi di risultato.

Capitolo 4

Le reti di Petri

4.1 Introduzione

Una rete di Petri, è una possibile rappresentazione matematica di un sistema distribuito discreto. Ideate da Carl Adam Petri, Matematico Tedesco, che le inventò nel 1962 durante la sua tesi di dottorato, vennero estese col passare degli anni e oggigiorno vengono utilizzate in moltissimi ambiti: nell'office automation, nei work-flows, nella produzione flessibile, nei linguaggi di programmazione, nei protocolli, nelle reti, nelle strutture hardware, nei sistemi real-time, nella valutazione delle performance, nelle operazioni di ricerca, nei sistemi embedded, nei sistemi di difesa, nelle telecomunicazioni, in internet, nell'e-commerce, nel trading, nelle reti ferroviarie e nei sistemi biologici.

Una rete di Petri, come un linguaggio di modellazione, descrive la struttura del sistema, rappresentandolo tramite un grafo bipartito diretto, in cui i nodi rappresentano le transizioni e i posti. Una transizione può rappresentare, ad esempio, un evento che può verificarsi nel sistema e viene indicata da una barra rettangolare; invece un posto può, ad esempio, rappresentare una condizione legata ad un evento e viene indicato con un cerchio vuoto. Un arco diretto indica quali posti sono pre-condizioni o post-condizioni di una certa transizione, il verso è indicato dalla freccia.

Quindi le reti di Petri offrono una notazione grafica per processi graduali che includono scelte, iterazioni ed esecuzione concorrente; a differenza degli altri standards industriali, come i diagrammi di attività UML, il BPMN e l'EPCs, le reti di Petri hanno una definizione matematicamente esatta della propria semantica di esecuzione, così come una teoria matematicamente ben sviluppata per quanto riguarda l'analisi dei processi.

Le reti di Petri, infatti, vanno a risolvere alcuni dei problemi che non si poteva risolvere con gli standard industriali. Ad esempio un activity diagram non può avere degli stati espliciti e ciò può essere limitante in alcune scenari; mentre gli state models non consentono attività concorrenti, ma in alcuni casi la concorrenza è desiderata o inevitabile. Le reti di Petri, invece, sono allo stesso tempo sia state-oriented che action-oriented, fornendo allo stesso tempo una descrizione degli stati del sistema e delle azioni che vengono eseguite su di esso.

Esistono 4 tipi diversi di reti di Petri:

- Ordinary nets: reti che si focalizzano sugli aspetti della concorrenza e della sincronizzazione, fornendo un ottimo strumento per costruire dei control models.
- Colored nets: reti che possono creare modelli più compatti e rappresentarne anche gli aspetti funzionali.
- Time-dependent nets: reti che permettono all'analista di inserire dei vincoli temporali e studiarne gli effetti sul sistema.
- Operational nets: reti che forniscono un linguaggio di alto livello per simulaizoni di sistema e sviluppo software.

Prima di esaminare ognuna di queste tipologie di reti, è necessario comprendere il significato degli elementi costitutivi della rete e delle proprietà comportamentali.

4.2 Elementi costitutivi

Una rete di Petri si compone di posti, transizioni e archi, infatti viene anche chiamata rete posto/transizione o più semplicemente rete P/T; questi 3 elementi definiscono la struttura della rete. Gli archi possono collegare un posto ad una transizione e viceversa, ma non possono mai collegare due posti o due transizioni. I posti con un arco uscente verso una transizione vengono detti input places della transizione e costituiscono il suo preSet, invece i posti con un arco entrante proveniente da una transizione vengono detti output places della transizione e costituiscono il suo postSet (viceversa, le transazioni entranti nel posto costituiscono il preSet del posto, mentre quelle uscenti il postSet) .
Matematicamente, una rete di Petri è definita da una struttura N e una sua situazione iniziale detta initial marking M_0. La struttura N è formata da una 4-tupla:

$$N=(P, T, F, W)$$

dove:

- $P=\{p_1, p_2, p_3,\ldots, p_{k-1}, p_k\}$; rappresenta l'insieme non vuoto dei posti.
- $T=\{t_1, t_2, t_3,\ldots, t_{m-1}, t_m\}$; rappresenta l'insieme non vuoto delle transizioni e deve essere disgiunto rispetto a P.

- F è un sottoinsieme di (P x T) unione di (T x P); rappresenta l'insieme non vuoti degli archi. F sta per Flow, infatti gli archi definiscono il flusso della rete.
- $W : F \rightarrow N^+$; rappresenta una weight function ossia una funzione che ad ogni arco della rete associa un numero naturale positivo che indica il peso dell'arco. Se il peso è pari ad 1, si può omettere.

L'initial marking M_0 invece rappresenta la situazione iniziale, indica in quali posti e quantità, all'istante iniziale t_0, sono presenti dei tokens. Se, ad esempio, all'istante t_0 avessimo 2 token nel posto p_2 e 0 in tutti gli altri, la marcatura iniziale sarebbe $M_0=\{0, 2, 0, \dots , 0, 0\}$.

I tokens sono il quarto elemento di una rete di Petri e ne definiscono il comportamento: I posti possono contenere i tokens; le transizioni sono le attività token-driven che hanno bisogno dei tokens per poter scoccare ed essere eseguite; gli archi definiscono le relazioni causa-effetto tra i tokens e le transizioni. La posizione dei tokens nella rete determina le attività (transizioni) che verranno eseguite e il loro ordine; ma una transizione per poter scoccare dev'essere prima abilitata. Per verificare se una transizione è abilitata, bisogna andare a guardare il numero di token presenti nei posti di input della transizione.

Nel caso in cui il peso degli archi sia pari ad 1, gli input places devono contenere almeno un token poichè quando la transizione scocca, rimuove un token da ogni posto di input e lo aggiunge in ogni posto di output. Questa regola è purtroppo non deterministica, infatti in caso ci siano 2 o più transizioni abilitate nel medesimo istante, essa non specifica quale transizione scoccherà per prima (viene considerata come un'operazione atomica e istantanea). Infatti, A meno che non siano state definite delle policies di esecuzione, in una rete di Petri l'esecuzione è non deterministica e quindi questo la rende adatta a rappresentare i modelli concorrenti dei sistemi distribuiti.

Nel caso in cui il peso degli archi sia diverso da 1, la transizione può scoccare solo se tutti i suoi posti di input hanno un numero di tokens maggiore o uguale al peso dell'arco che li congiunge alla transizione. Questo implica che archi diversi che congiungono posti diversi alla stessa transizione possono avere pesi diversi e quando scossa, la transizione rimuove dal un numero di tokens pari al peso dell'arco e aggiunge nei posti uscenti un numero di tokens pari al peso degli archi uscenti dalla transizione.

Dopo aver posizionato i vari tokens nei posti di output, il sistema aggiorna la marcatura corrente: essa viene solitamente implementata attraverso un vettore M in cui ogni elemento M[i] contiene il numero di tokens presenti nel posto p_i. Pertanto, il comportamento di una rete di Petri è dato dall'evoluzione della sua marcatura nel tempo e viene determinato dalla marcatura iniziale (initial marking) e il grafo di raggiungibilità mostra tutte le possibili marcature e l'ordine di attivazione

delle transizioni. Il grafo di raggiungibilità, o reachability graph, è costruibile solo se la rete è limitata, ossia i suoi posti contengono un numero limitato di tokens.

Queste regole consentono l'uso di pattern per il control flow, ossia il controllo del flusso: il flusso può essere diviso da una transizione di tipo fork, in cui dato un di arco entrante vi sono 2 o più archi uscenti; il flusso può essere riunito da una transazione di tipo join, in cui dati almeno 2 archi entranti vi è un solo arco uscente dalla transizione; ma si possono avere anche entrambe, ad esempio se abbiamo una join con 2 archi entranti e una fork con 3 archi uscenti dalla stessa transizione. Se invece avessimo un posto P con un arco entrante(quindi una transizione) e 2 archi uscenti (quindi 2 transizioni) le cui transizioni hanno (tutte) solo il posto P come posto di input, allora la scelta su quale arco immettere i tokens non può essere presa automaticamente con gli elementi che abbiamo visto finora, poichè l'esecuzione è di tipo non deterministico; questa particolare configurazione prende infatti il nome di free-choice, ovvero scelta libera. Oltre alla free-choice, vi sono altre 2 situazioni in cui la decisione non può essere presa in maniera automatica a causa di un conflitto:

1. Dati due posti p_1 e p_2, il primo ha due transizioni uscenti t_1 e t_2, il secondo ha anche lui 2 transizioni uscenti t_2 e t_3. Questa situazione viene chiamata symmetric confusion, ossia confusione simmetrica, perché la transizione t_2 ha più posti di input ed essi hanno almeno un'altra transizione uscente non in comune tra loro. In questo caso se ci fosse un token solamente in p_1, scatterebbe t_1; se ci fosse un token solamente in p_2, scatterebbe t_3; invece se ci fosse un token in p_1 e un token in p_2, tutte e 3 le transizioni sarebbero abilitate, ma se scoccasse t_2 ruberebbe un token da entrambi i posti, bloccando le altre due transizioni e quindi c'è un conflitto nella decisione.
2. Dati due posti p_1 e p_2, il primo ha due transizioni uscenti t_1 e t_2, il secondo ha solamente la transizione uscenti t_2. Questa situazione viene chiamata asymmetric confusion o asymmetric choice, ossia confusione asimmetrica o scelta asimmetrica, perché la transizione t_2 ha più posti di input e tra questi vi è almeno un posto in cui essa è l'unica transizione uscente e vi è almeno un altro posto che, oltre a t_2, ha un altra transizione uscente. In questo caso se ci fosse solamente un token in p_1, scatterebbe t_1; se vi fosse un token solamente in p_2, non scatterebbe nessuna transizione (manca il token in p_1); invece se si fosse un token in entrambi i posti, ci sarebbe un conflitto nella decisione, perché lo scoccare dell'una precluderebbe lo scoccare dell'altra.

4.3 Regole e proprietà comportamentali

Il grafo di raggiungibilità può fornirci importanti informazioni sulla rete e sulle sue proprietà comportamentali:

- Se ogni posto non conterrà più di un token alla volta, avremo una rete safe. La proprietà di Safeness è in realtà una sotto-proprietà della proprietà di Boundedness, per cui la rete si dice limitata di un fattore k se il numero di tokens presenti in ogni posto non supera il fattore k.
- Se ogni marcatura abilita almeno una transizione, avremo una rete deadlock-free. La proprietà di Deadlock-freedom implica che l'esecuzione della rete di Petri arriva ad un punto morto, detto deadlock, in cui l'esecuzione non può procedere perché nessuna transizione può scoccare. Questo si verifica quando si arriva al punto in cui 2 o più transizioni sono in attesa di risorse reciprocamente oppure in corrispondenza di una free-choice.
- Se per ogni transizione si può sempre riottenere una marcatura in cui la transizione è abilitata, avremo una rete live. La proprietà di Liveness implica che ogni transizione della rete possa sempre scattare e quindi la rete rimane viva. Più nel dettaglio, una transizione T è live se e solo se, per ogni marcatura M_i esiste una marcatura M_j raggiungibile da M_i, tale per cui T è abilitata in M_j; al contrario T si dice morta (dead) se non viene più abilitata in nessuna marcatura raggiungibile da M_i. Liveness e Deadlock-freedom non sono totalmente indipendenti l'una dall'altra, infatti se una rete è live allora è anche deadlock-free e se non è deadlock-free allora non è live; invece se una rete è deadlock-free non possiamo trarre alcuna conclusione sulla sua liveness; discorso analogo nel caso ci sia una rete non live: non possiamo trarre alcuna conclusione sulla sua Deadlock-freedom.
- Se la marcatura iniziale si può sempre riottenere, avremo una rete reversible. La proprietà di reversibility implica che per ogni marcatura M_i, è sempre possibile raggiungere la marcatura iniziale M_0 e, in generale, ogni marcatura raggiungibile da un'altra marcatura viene detta home-state e la sua presenza implica la presenza di un sistema periodico.

Le proprietà di Boundedness, Liveness e Reversibility sono indipendenti tra di loro; mentre la proprietà di Deadlock-freedom può essere vista come una proprietà più debole rispetto alla Liveness.

Il grafo di raggiungibilità può essere calcolato solamente nelle reti di Petri limitate (bounded), in caso di rete illimitata (unbounded) dobbiamo utilizzare un altro strumento per ottenere le

informazioni necessarie allo studio della rete. Inoltre, la costruzione del grafo di raggiungibilità è onerosa poiché bisogna numerare tutte le marcature e quindi adatto solo a reti piccole. In questi casi (rete illimitata o non piccola), si può sfruttare il coverability graph che è un grafo approssimativo che mostra tutte le marcature coverable, ossia ricopribili, in cui quando il numero di tokens presenti in un posto aumenta tendendo all'infinito, allora nel vettore delle marcature si inserirà una ω che simboleggia un numero di tokens infinito.

Pertanto, dato un reachability graph e una sua marcatura M, essa è raggiungibile se esiste un nodo che la contiene; invece dato un coverabilty graph e una sua marcatura M, non possiamo trarre conclusioni sulla raggiungibilità, ma possiamo solamente dire che M è coverable se esiste un nodo contenente M_i che ricopre M. Inoltre, attraverso un coverabilty graph non possiamo ottenere informazioni neanche sulla Liveness poiché, essendo un grafo approssimativo, possono esistere due reti con lo stesso grafo, ma con diversa liveness (una live e l'altra no).

4.4 Tipi di reti

La costruzione del grafo è un'operazione onerosa, per questo motivo si preferisce studiare le reti di Petri focalizzandosi sulla sottoclasse di appartenenza; infatti, sono state definite delle sottoclassi di reti di Petri in base alla tipologia della rete che offrono la possibilità di uno studio più mirato e meno oneroso. Le dipendenze tra le varie sottoclassi sono di tipo gerarchico: la classe principale è rappresentata dalle Petri Nets (PNs); da essa deriva la sottoclasse delle Asymmetric Choice Nets (ACs); da questa derivano le Extended Free-Choice Nets (EFCs); all'interno delle EFCs abbiamo le Free-Choice Nets (FCs); infine all'interno delle FCs possiamo trovare State Machine Nets (SMs) e Marked Graph Nets (Mgs).

Nelle reti di tipo State Machine (macchina a stati) ogni transizione ha esattamente un posto di input e un posto di output, sono ammesse le Free-choice (le SMs sono una stottoclasse delle FCs), ma non sono ammesse le transizioni di tipo fork e join. Se in una State Machine Net, il marking iniziale contiene un solo token, la SM è equivalente ad uno state-transition diagram (diagramma degli stati) senza eventi e senza azioni. Una rete di tipo SM è live se e solo se la sua marcatura iniziale contiene almeno un token ed è safe se contiene esattamente un token.

Nelle reti di tipo Marked Graph (grafo marcato) ogni posto ha esattamente una transizione di input e una transazione di output e possono essere usati per rappresentare reti concorrenti e sincronizzate, ma non possono rappresentare le scelte perché due transizioni non possono avere un posto in

comune. Non essendoci né scelte, né confluenze, le proprietà di un MG possono essere studiate utilizzando i circuiti: una sequenza di transizione e di posti visti come elementi distinti di un circuito tali per cui ogni elemento fa parte del postSet dell'elemento precedente e il primo elemento della sequenza fa parte del postSet dell'ultimo elemento (il circuito si chiude). Pertanto in un MG il numero totale di tokens (il tokens count) in ogni circuito diretto non cambia nel tempo ed è quindi calcolabile basandosi sulla marcatura iniziale. Di conseguenza, una rete MG è live se il token count di ogni circuito diretto è maggiore di 0 ed è safe se ogni posto appartenente ad un circuito diretto ha token count pari a 0; il numero massimo di tokens che può contenere un posto è pari al token count più basso tra quelli dei circuiti a cui appartiene; una rete MG è bounded se e solo se è fortemente connessa e in tale caso esiste sicuramente una marcatura live e safe.

Nelle reti di tipo Free-Choice (scelta simmetrica), ogni posto che ha almeno 2 transizioni di output è l'unico posto di input per tali transizioni; questo tipo di rete può rappresentare sia reti sincronizzate, sia reti con conflitti semplici (free-choices). Le proprietà di una FC possono essere determinate sfruttando i concetti di sifone e trappola. Un sifone è un sottoinsieme non vuoto dei posti della rete, in cui ogni transizione che ha un posto di output in tale sottoinsieme, ha anche un posto di input in quello stesso sottoinsieme. Pertanto ogni transizione che inserisce un token in un sifone, ne prende anche uno da esso e quindi un sifone che è vuoto in una certa marcatura, sarà vuoto in tutte le marcature successive. Una trappola, invece, è un sottoinsieme non vuoto dei posti della rete, in cui ogni transizione che ha un posto di input in tale sottoinsieme, ha anche un posto di output nello stesso sottoinsieme. Pertanto ogni transizione che prende un token da una trappola, ne mette anche un altro e quindi se una trappola avrà un token ad una certa marcatura, avrà quel token anche in tutte le marcature successive. Unendo più sifoni otteniamo un altro sifone, e unendo più trappole otteniamo un'altra trappola. Se consideriamo tutti i posti della rete, questo insieme è sia sifone che trappola. Infine possiamo ancora dire che una rete FC è live se e solo se i suoi sifoni contengono una trappola marcata inizialmente (teorema di Commoner) e questa è una ricerca esponenziale.

Nelle reti Extended Free-Choice (scelta libera estesa), ogni 2 posti che hanno una transizione di output in comune devono avere anche tutte le altre transizioni di output in comune e rappresentano un estensione delle FC. Infatti, poiché hanno in comune tutte le transizioni di output, se una transazione è abilitata, lo sono anche le altre in comune. Con opportune trasformazioni, una EFC può essere trasformata nella rispettiva FC.

Nelle reti Asymmetric choice (scelta simmetrica), se 2 posti hanno in comune una transizione di output, allora l'insieme delle transizioni di output di un posto è un sottoinsieme delle transizioni di output dell'altro o viceversa. Le reti AC possono rappresentare la confusione asimmetrica e sono

live se e solo se ognuno dei suoi sifoni contiene una trappola marcata inizialmente (teorema di Commoner).

Per valutare le performance di una rete di Petri, si sfrutta la simulazione a eventi discreti (di cui abbiamo abbondantemente parlato nel primo capitolo), inserendo dei ritardi (delays) nelle transizioni che quindi se ricevono un token al tempo t, lo restituiranno al tempo t+d dove d rappresenta il ritardo. Ad esempio, un MG fortemente connesso, in cui vengono inseriti i ritardi, viene detto Timed Marked Graph (TMG) e da esso si possono ottenere dei risultati analitici.

4.5 Reti di Petri object-oriented

Le reti di Petri object-oriented sono più complesse rispetto alle reti di Petri ordinarie perché alla gestione della rete, va aggiunta la gestione degli oggetti e dei dati contenuti al loro interno (trattamento dati). In questo tipo di reti, che rientra sotto la categoria delle reti colorate (coloured nets), i tokens sono dei riferimenti a degli oggetti che contengono dei dati e le transizioni manipolano questi oggetti attraverso i tokens. Un oggetto posto può contenere molti oggetti tokens, tutti dello stesso tipo, ed essi vengono inseriti in una coda ordinata per ordine di arrivo; inoltre, un posto contiene 2 stringhe, la prima indica il nome del posto, la seconda il tipo di tokens che può contenere. Nel caso in cui un posto contenesse tokens vuoti, è possibile omettere il nome della classe del token.

Il Data Flow (flusso dei dati) si ottiene studiando gli oggetti trasportati dai tokens e questo fa si che il Data Flow e il Control Flow siano combinati. Il trasporto dei tokens è affidato alle transizioni che sono anche l'unità di processing del modello: esse contengono una guardia (indicata con g) e una azione (indicata con a). La guardia seleziona dai posti di input i tokens necessari a far scattare la transizione: l'azione effettua il processing dei dati trasportati dai tokens e li inserisce nei posti di output della transizione.

La guardia, quando definita, viene detta guardia esplicita (explicit guard) ed è formata da una condizione booleana utilizzata per selezionare i tokens dai posti di input della transizione in base al loro contenuto: se esso soddisfa la condizione, il token verrà selezionato e la transizione verrà abilitata, altrimenti, verrà ignorato e la transizione non verrà abilitata; questo permette una selezione più precisa rispetto alla regola ordinaria in cui veniva selezionato il primo token della coda che corrisponde al token più vecchio. Ai tokens vengono dati dei nomi che richiamano i posti da cui provengono e se nella condizione non viene menzionato il posto, si prende semplicemente il primo token della coda. Se invece la guardia viene omessa, per selezionare i tokens si utilizza la regola di

default, cioè quella ordinaria. Questo controllo sulla condizione della guardia viene effettuato ogni volta che un token viene inserito in un posto di input della transizione; se più tokens dello stesso posto soddisfano la condizione, si può inserire una clausola del tipo order by per ordinare i tokens in modo che la selezione avvenga correttamente.

Poichè, quando ci sono 2 o più transizioni abilitate, non è possibile stabilire quale transizione scatterà per prima, si assegna ad ogni transizione una priorità: essa viene rappresentata da un intero maggiore o uguale a 0; la priorità di default è pari a 0, e solitamente corrisponde al numero scritto nell'attributo name dopo il nome della transizione. Se vi sono 2 transizioni abilitate con la stessa priorità, il sistema deve scegliere quale far scoccare per prima.

Abbiamo visto che ogni token di un posto di input viene inviato ad un posto di output che ammette lo stesso tipo di tokens, altrimenti il token viene scartato; infatti, se il tipo ammesso è diverso, verrà creato un nuovo token dalla transizione che verrà inserito nel posto di output, mentre il vecchio token verrà scartato.

L'ambiente di simulazione che abbiamo creato sfrutta proprio queste regole durante l'esecuzione della rete di Petri associata al modello studiato.

Capitolo 5

Processi di business B2B

5.1 Introduzione

I processi di business B2B sono dei processi di business in cui più organizzazioni interagiscono tra di loro attraverso lo scambio di messaggi, grazie a task specifici che mandano o ricevono i messaggi. In questo caso, si parla di Entity-Oriented Nets, ossia reti di Petri basate sul concetto di entità: i tokens si riferiscono ad un'entità del sistema informativo che viene costruito sulla base del modello informativo del processo in cui sono presenti le informazioni su tutte le entità coinvolte nel processo.

Le reti entity-oriented sono simili alle reti object-oriented, ma pongono maggiore enfasi sulle entità coinvolte nel processo e suoi ruoli dei partecipanti:

- I posti hanno anche qui 2 etichette: la prima indica il tipo di entità che possono contenere, la seconda indica il nome dello stato di quel posto nel ciclo di vita (life cycle) delle entità.
- Le transizioni rappresentano i task: essi possono essere automatici, ossia eseguiti dal sistema senza l'intervento umano, oppure possono essere umani (human tasks), in cui è richiesta l'azione di uno dei partecipanti, ossia di una delle persone coinvolte nel processo che hanno il proprio ruolo e task associati. Queste persone, non sono semplici esecutori, ma possono scegliere quale task eseguire per elaborare i dati e quali dati fare elaborare da un task. I task vengono suddivisi in categorie ed essi elaborano i dati ricevuti; gli effetti che ne derivano vengono chiamati post-condizioni, mentre i vincoli associati ad un task che bloccano l'esecuzione del task vengono chiamati pre-condizioni. Possono essere creati anche task composti, ossia dei task che contengono al loro interno un insieme di altri task e l'esecuzione del task composto implica l'esecuzione dei task contenuti al suo interno.

Le reti entity-oriented possono essere applicate a due scenari di processi di business. Il primo riguarda i processi a singola piattaforma o a piattaforma comune, in cui un'organizzazione interagisce con i suoi partners e i ruoli sono affidati sia a persone che fanno parte dell'organizzazione, quindi chiamati ruoli interni, sia ai partners, quindi chiamati ruoli esterni. Il secondo scenario è quello dei processi B2B, in cui non esiste alcuna piattaforma comune, ma

l’organizzazione e i partners interagiscono attraverso scambi di messaggi regolati da protocolli di comunicazione ben precisi; infatti si parla di scambio asincrono. Pertanto, per i processi B2B, non basta il modello informativo, ma è necessario definire anche un modello collaborativo.

L’analisi completa di un processo di business B2B richiede quindi la definizione di un modello informativo (entità contestuali, entità di processo e invarianti), un modello collaborativo (protocolli di collaborazione) e la definizione del processo attraverso il process model. Infatti, in un B2B il sistema è composto da un numero di organizzazioni che collaborano per raggiungere uno scopo comune. Queste organizzazioni sono associate ad un ruolo (acquirente, venditore, fornitore, ecc..) e collaborano attraverso i propri processi che inviano e ricevono messaggi secondo i protocolli di collaborazione. Il modello collaborativo stabilisce quali messaggi devono essere scambiati tra le 2 parti al fine di raggiungere l’obiettivo prefissato; questo modello di comunicazione binaria va definito prima dell’implementazione vera e propria del processo di business per evitare malfunzionamenti nel processo.

I modelli vengono spesso utilizzati per studiare le proprietà di un sistema complesso: essi sono delle rappresentazioni astratte e accurate del sistema che consentono l’analisi attraverso un insieme di proprietà ben definite. Nel B2B, il modello descrive il contesto nel quale opererà il sistema e le relazioni con gli altri sottosistemi dal punto di vista aziendale ed è quindi un ponte tra gli analisti di business e gli analisti software.

Per fare in modo che il modello sia comprensibile a tutti gli addetti del settore, è necessario utilizzare un linguaggio semplice e universale nel descrivere le varie componenti del sistema; a tale scopo è stato scelto UML (Unified Modeling Language) che comprende un insieme di linguaggi e notazioni utili per costruire i modelli. Esso si appoggia allo standard OMG e fornisce principalmente 2 tipi di diagrammi: Structural Diagram, ossia il diagramma strutturale che contiene le classi e gli oggetti coinvolti nel processo; Behavioral Diagram, ossia il diagramma comportamentale che contiene le attività, gli stati, le sequenze e i casi d’uso. Per determinare i vincoli legati agli oggetti, solitamente si usa OCL (Object Constraint Language): un linguaggio UML formale usato come i linguaggi di query, per specificare invarianti, pre-condizioni e post-condizioni Essendo un linguaggio specificativo, OCL non produce effetti collaterali, infatti, quando viene valutata un’espressione OCL, l’operazione ritorna solamente un valore, senza andare a modificare in nessun modo i dati valutati e quindi lasciando inalterato il modello.

5.2 Modello informativo

Per definire il modello informativo è necessario individuare tutti gli elementi presenti nel sistema:

- Ad ogni elemento corrisponde un oggetto chiamato object.
- Ogni oggetto appartiene ad una classe detta class.
- Ogni collegamento tra due oggetti viene rappresentato da un link detto relationship che connette le classi di appartenenza dei due oggetti.
- Le classi possono avere delle proprietà chiamate attributes.

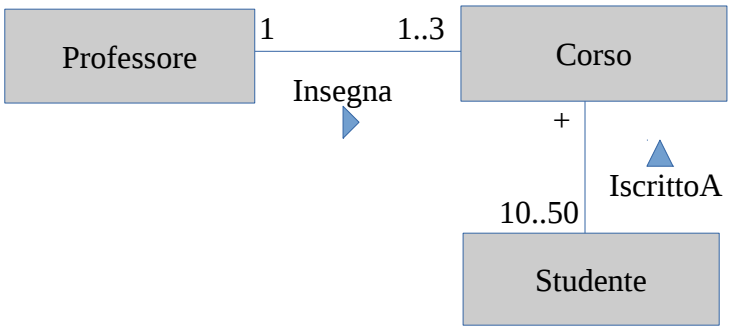
Definendo questi elementi è possibile costruire il Domain Model del processo che mostra le classi, gli attributi e le relazioni. Il Domain Model indica il dominio di applicazione ed è un modello che descrive il sistema da un punto di vista di alto livello come quello aziendale e concettuale, ma esso è sinonimo di modello informativo (information model) e modello delle classi (class model): il primo usato quando si studia il sistema dal punto di vista della base dati; il secondo dal punto di vista tecnico e quindi dell'implementazione del sistema. E' infine possibile costruire anche un object model, ossia un class model particolare in cui vengono mostrati gli oggetti correlati del sistema. Ad esempio, se consideriamo un'università, essa è composta da professori e studenti; i professori insegnano diversi corsi e gli studenti sono iscritti a tali corsi.

Il primo passo è quello di identificare le classi presenti nel sistema e rappresentarle attraverso un rettangolo con all'interno il nome della classe; in questo caso nel nostro modello informativo abbiamo: Professore, Corso, Studente; quindi gli oggetti saranno le istanze di tali classi.

Il secondo passo consiste nell'esaminare le relazioni tra le varie classi del sistema e rappresentarle attraverso i links (una linea che congiunge le 2 classi), tenendo presente che sono ammesse solo relazioni associative binarie: Professore e Corso hanno una relazione (ogni professore insegna un corso), quindi aggiungiamo un link con il nome "Insegna"; Professore e Studente non hanno apparentemente alcuna relazione, quindi non aggiungiamo nulla; Corso e Studente hanno una relazione (ogni studente è iscritto ad un corso), quindi aggiungiamo un link con il nome "Iscritto a".

Il passo finale è quello di aggiungere la molteplicità alle relazioni individuate in precedenza, in questo modo è anche possibile specificare eventuali vincoli: poiché un corso è solitamente tenuto da 1 solo professore, si aggiunge sul collegamento Professore-Corso, dal lato Professore, un numero pari ad 1; mentre, poiché ogni professore può insegnare da 1 a 3 corsi, si aggiunge sullo stesso collegamento, ma sul lato Corso, l'espressione "1..3" che indica una molteplicità da 1 a 3. Notare

che questo implica anche che non può esistere un corso senza professore (molteplicità minima pari ad 1), non può esistere un professore senza corso (molteplicità minima pari ad 1) e egli non può insegnare più di 3 corsi (molteplicità massima pari a 3). Analogamente, nella relazione "Iscritto a" abbiamo i seguenti vincoli: poichè ogni studente è iscritto almeno ad un corso e non c'è un limite al numero di corsi a cui uno studente può iscriversi, si aggiunge un "+" sul lato Corso che indica una molteplicità da 1 a infinito; mentre, poiché ogni corso può avere da 10 a 50 studenti, si aggiunge l'espressione "10..50" sul lato Studente del collegamento. Anche in questo caso, le molteplicità implicano che non possa esistere uno studente non iscritto a nessun corso e che dato un corso, non possono esserci né meno di 10 studenti iscritti né più di 50. Il modello informativo finale è il seguente:



Attraverso la molteplicità possiamo quindi specificare dei vincoli; nel caso in cui non ci siano vincoli in una relazione, viene assegnata la molteplicità di default "*" oppure "0..*" che indica un numero da 0 (compreso) a infinito; se infine vogliamo definire una relazione opzionale, possiamo usare l'espressione "0..1".

Esistono anche relazioni particolari con molteplicità definita: una fra queste è la composizione di oggetti che viene utilizzata quando un oggetto è composto da altri oggetti del modello. Di conseguenza distruggendo un oggetto composto verranno distrutti tutti gli oggetti che lo compongono, detti componenti; invece una componente non può esistere senza l'oggetto composto a cui appartiene, detto contenitore. Ad esempio, se consideriamo uno scontrino, esso è composto da una lista di prodotti e quantità acquistate; in questo caso, nel modello informativo, useremo la notazione di composizione, ossia un rombo pieno sul collegamento dal lato della classe contenitore, mentre sul lato della classe componente si aggiungerà una "n" che indica una molteplicità maggiore di 1. Per rappresentare la quantità venduta, viene semplicemente aggiunto un attributo Integer nella classe Prodotto. La composizione è un link di tipo forte, infatti un componente può appartenere ad un solo contenitore, se invece è necessario che un componente abbia più di un contenitore, si utilizza la notazione di aggregazione che viene invece indicata con un rombo vuoto. In questo caso,

l’eliminazione dell’oggetto contenitore non implica più l’eliminazione delle sue componenti e di fatto questo tipo di relazione può essere sostituita da una relazione associativa.

Un altro tipo di relazione è l’associazione tra classi: essa permette di associare degli attributi ad una relazione sfruttando un pattern associativo in cui si inserisce una classe in mezzo alla relazione fra altre due classi. Ad esempio, consideriamo nuovamente l’oggetto Scontrino, ma stavolta vogliamo avere degli oggetti Prodotto indipendenti dagli oggetti Scontrino, ma tenere ugualmente conto del numero di prodotti acquistati; per fare questo possiamo sfruttare l’associazione tra classi e inserire tra le due classi una classe Linea che contiene un riferimento ad un oggetto Prodotto, un riferimento ad un oggetto Scontrino e un attributo quantità. Per quanto riguarda la molteplicità, essa sarà pari ad “1”, lato Scontrino, nel link Scontrino-Linea e sarà pari a “*” lato Linea; mentre nel link Linea-Prodotto sarà “*” lato Linea e “1” lato Prodotto.

Un altro tipo di relazione è l’Inheritance, ossia l’ereditarietà: molto utile quando un oggetto deve ereditare le proprietà di un altro oggetto al fine di estenderlo. Ad esempio, se consideriamo una classe astratta Veicolo, questa può essere estesa ereditando le proprietà della classe madre e aggiungendo delle proprietà specifiche delle classi derivate. Possibili estensioni sono Auto, Bus, Moto, Tir ecc.. e avranno una relazione di ereditarietà verso Veicolo, rappresentata da un triangolo vuoto, inserito sul link a ventaglio che collega le classi, sul lato Veicolo; mentre sul lato delle classi derivati non inseriamo nulla perché è sottintesa la molteplicità di default “*”.

Dal punto di vista tecnico, il modello informativo corrisponde al modello delle classi e questo può essere interpretato in due modi:

- Dal punto di vista programming-oriented, ossia orientato alla programmazione: il modello rappresenta un programma (ad esempio un programma java)e le classi del modello vengono mappate sulle classi java; gli attributi delle classi del modello vengono mappati sugli attributi delle classi java; le relazioni di ereditarietà vengono mappate sfruttando la parola chiave “extends”; tutte le altre relazioni vengono mappate attraverso dei riferimenti o delle liste.
- Dal punto di vista information-oriented, ossia orientato all’informazione contenuta negli oggetti: il modello segue un approccio object-relational (relazionale ad oggetti) e rappresenta un’informazione persistente salvata in un database, a cui si può accedere attraverso gli oggetti java. Dal modello è quindi possibile definire le tabelle relazionali e le classi java i cui oggetti rappresentano i records del database; questi oggetti vengono detti entities (entità) e le loro classi vengono dette entity classes; di conseguenza, il class model viene anche detto entity model. E’ possibile effettuare 6 tipi di operazione sulle Entità: 4 di

queste sono le così dette operazioni CRUD (Create, Read, Update, Delete) usate per la creazione, la lettura, l'aggiornamento e la cancellazione dei dati nei database; le altre 2 servono per aggiungere o rimuovere un associazione tra 2 classi nel database.

5.3 Modello Collaborativo

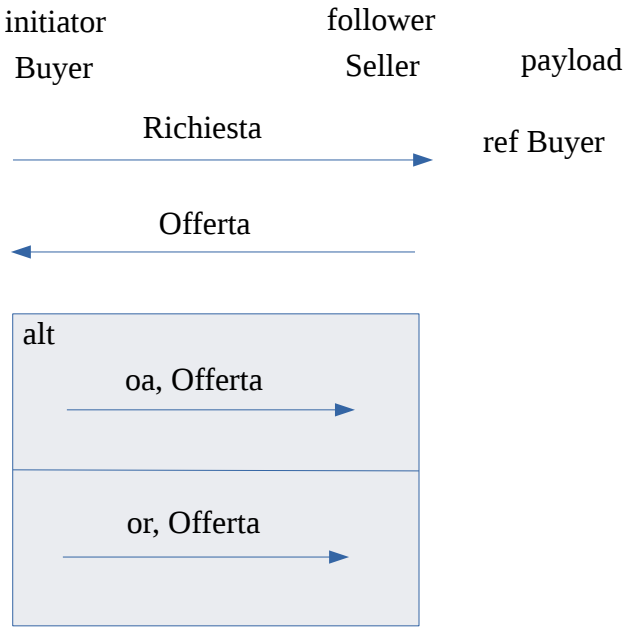
Le organizzazioni collaborano attraverso dei processi, nei quali interagiscono inviando e ricevendo dei messaggi che rispettano protocollo predefiniti, rappresentabili attraverso un modello collaborativo. Esso stabilisce quali messaggi vengono scambiati dalle due parti per raggiungere lo scopo prefissato; infatti si tratta di una comunicazione binaria. Questi messaggi sono strettamente correlati alle business entities, ossia le entità coinvolte nel processo di business: infatti, un messaggio uscente conterrà un'entità specifica, corrispondente ad un oggetto del sistema informativo del mittente; mentre da un messaggio entrante verranno ricavate le informazioni necessarie per generare o aggiornare un'entità specifica nel sistema informativo del ricevente. Di fatto, la comunicazione nelle collaborazioni B2B consiste nell'allineare i sistemi informativi di tutti i partners coinvolti, nonostante essi abbiano sistemi informativi distinti. Questo avviene grazie alla definizione un numero di entità condivise chiamate global entitites, ossia entità globali, conosciute da tutti i partners; per questa ragione, nei messaggi si fa riferimento a queste entità.

Quindi, da un punto di vista concettuale, le interazioni trasportano le entità, con gli attributi indicati nel modello di collaborazione, mediante messaggi. Ogni messaggio include un header, in cui ci sono le informazioni utili per interpretare il messaggio, e un payload in cui è contenuta un'entità del tipo indicato nell'interazione. Ad esempio, consideriamo un caso in cui abbiamo due organizzazioni : un acquirente (un Buyer) e un venditore (un Seller). Vogliamo implementare un protocollo collaborativo richiesta-risposta: in questo caso il Buyer sarà l'initiator, ossia colui che inizia lo scambio di messaggi; mentre il Seller sarà il follower, ossia colui che segue e che risponde ai messaggi ricevuti. Quindi abbiamo 2 possibili interazioni: il Buyer che invia una Richiesta; il Seller che risponde ad una richiesta inviando un'Offerta. Nello specifico il seller, quando riceve una richiesta, genera automaticamente un'entità di tipo Richiesta con le informazioni contenute nel payload del messaggio; inoltre, poiché l'entità Richiesta è esogena per il seller (l'originale è stata creata dal Buyer e quindi al di fuori del sistema informativo del Seller), essa sarà collegata all'entità che ne rappresenta il mittente: Il seller quindi nel proprio sistema informativo avrà un'entità Buyer a cui collegherà la Richiesta ricevuta.

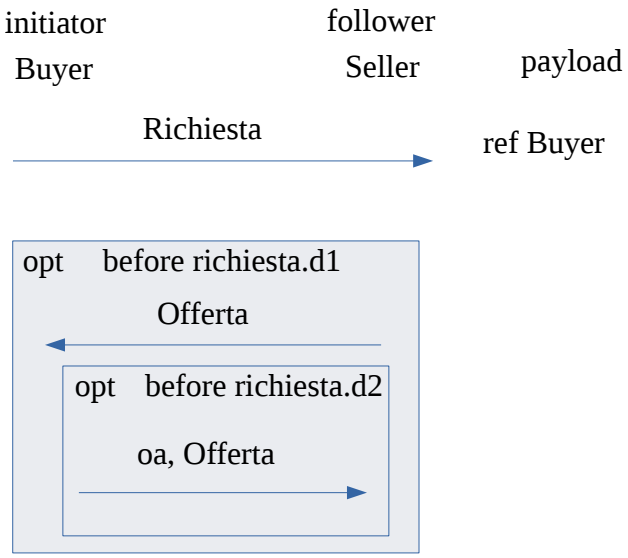
Il Buyer manda al Seller una Richiesta di offerta relativa ad un tipo di prodotto offerto da quel Seller; pertanto nell'entità Richiesta ci sarà un attributo di tipo String che conterrà la descrizione del prodotto richiesto. Una volta ricevuta la Richiesta, il Seller genererà un oggetto Richiesta con un riferimento al Buyer della richiesta; si parla infatti di Task generativo. A questo punto il Seller invierà al Buyer un messaggio contenente l'offerta proposta e questo genererà un'entità Offerta nel proprio sistema informativo (anche questo è un Task generativo); anche in questo caso abbiamo un attributo di tipo String che descrive l'offerta proposta. Il Buyer a questo punto può scegliere se accettare o rifiutare l'offerta del Seller (siamo nel caso di un'alternativa, indicabile con "alt") e, a seconda della scelta, invierà al Seller un messaggio contenente la stessa offerta ricevuta insieme alla decisione presa, ad esempio un attributo che può essere "oa" (offerta accettata) oppure "or" (offerta rifiutata); questo attributo da il nome all'offerta modificativa che andrà quindi a modificare l'oggetto già presente nel sistema informativo del Seller. Qualora volessimo una risposta opzionale ad un messaggio, sarebbe necessario aggiungere una deadline, ossia una scadenza, alla Richiesta; in modo che il partner non resti in attesa inutilmente nel caso in cui l'altro non volesse rispondere. A scadenza, se non è pervenuta alcuna risposta, vengono eliminate dai rispettivi sistemi informativi tutte le entità generate da quella collaborazione che viene considerata come terminata o chiusa.

Da un punto di vista astratto, un'interazione può essere vista come un evento creato da una delle due parti che interferisce con l'altra parte, attraverso le informazioni sull'entità (information entities) contenuta nel payload del messaggio. Pertanto i modelli collaborativi si basano sui diagrammi di sequenza UML e si parla di State Machine (macchina a stati, non c'è concorrenza). La sequenza di interazioni viene rappresentata graficamente con una disposizione verticale e le interazioni sono raggruppate in blocchi, detti frammenti. Il Control Flow (flusso di controllo) viene definito attraverso l'uso di costrutti di controllo: "opt" per indicare un blocco opzionale, "alt" per indicare un blocco in cui è presente una scelta, "loop" per indicare un blocco ciclico, "break" per uscire immediatamente da un blocco ciclico, "end" per indicare la fine del blocco che verrà ripetuto nel ciclo; se non viene indicato alcun costrutto, si assume il costrutto di default, ossia "opt". I diagrammi di sequenza vengono utilizzati soprattutto per scopi illustrativi e normativi.

Considerando l'esempio precedente Buyer-Seller il diagramma di sequenza (quindi il modello collaborativo), nel caso in cui il tempo di risposta sia illimitato, è il seguente:



In cui la collaborazione termina con oa (offerta accettata) oppure con or (offerta rifiutata). Se invece le risposte avessero delle scadenze, rispettivamente d1 e d2, il modello collaborativo sarebbe il seguente:



In cui la collaborazione termina con oa (come sopra), oppure con d1 (se non arriva alcuna offerta dopo la scadenza), oppure con d2 (se il task modificativo "oa" non viene eseguito e quindi l'offerta non viene accettata).

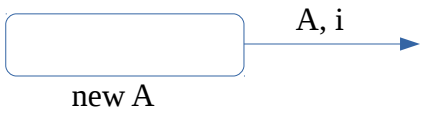
5.4 Process Model

Il process model è un modello che raggruppa un insieme di processi dello stesso tipo che ne costituiscono le istanze, esso è composto da una serie di task. Un process model viene usato ripetutamente durante lo sviluppo di molte applicazioni e ha molte istanze (processi) ed esso rappresenta un'anticipazione di come sarà il processo reale e come verrà definito durante la fase di sviluppo. Un process model ha 3 diversi scopi:

- Descrittivo: è possibile tracciare e verificare cosa accade realmente durante il processo e osservare il processo dal punto di vista di un osservatore esterno che ,analizzando il modo in cui esso viene eseguito, può proporre dei miglioramenti per aumentare l'efficienza del processo.
- Prescrittivo: Permette di definire i processi e il modo in cui essi devono essere eseguiti, stabilendo delle regole, delle linee guida e pattern comportamentali che avranno come risultato il processo desiderato. Sono possibili diverse modalità, dai vincoli restrittivi a linee guida flessibili.
- Esplicativo: Fornisce una spiegazione riguardo alla logica del processo, permette di esplorare e valutare tutte le possibili sequenze di azioni basandosi su argomenti razionali, stabilisce un collegamento esplicito tra i processi e i requisiti che il modello deve soddisfare, stabilisce a priori i punti in cui i dati possono essere estratti per i reports.

Questi modelli vengono spesso utilizzati per identificare possibili problemi nei processi, per correggerli o semplicemente quando bisogna modificare una parte di un processo e in linea teorica, questa trasformazione non necessita il coinvolgimento di un esperto IT, proprio perché il linguaggio utilizzato è semplice e di facile comprensione. Di seguito andremo ad esaminare i pattern principali utilizzati nel process model.

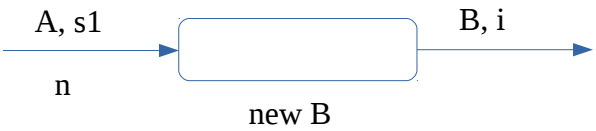
Il primo pattern corrisponde, solitamente, ad un task di ingresso: non ha nessun input, ma ha una post-condizione che implica la creazione di una nuova entità (è anche un task generativo) e quindi utilizza il costrutto "new"; l'entità prodotta costituisce l'output del task a cui si associa lo stato corrente del dato emesso. Se ad esempio l'entità prodotta fosse A, il suo stato corrisponde allo stato iniziale "i":



Un altro pattern corrisponde ad un task di uscita: non produce alcun output, non ha nessuna condizione, ma ha solamente l'input: ad esempio, considerando nuovamente un'entità A, il suo input sarà dato da A e dal suo stato corrente (ad esempio "s1"):

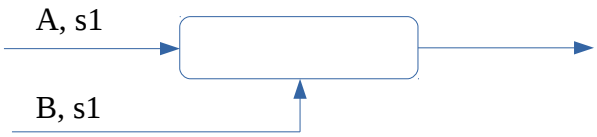
Un altro tipo di pattern corrisponde ad un task modificativo: l'entità di input e di output coincide, ma varia lo stato dell'entità; di fatto, il task attua una modifica sull'entità che verrà inviata al task successivo con un nuovo stato (ad esempio "s2"):

Un altro pattern corrisponde ad un task aggregativo/costruttivo: riceve un numero n di entità in input e, una volta ricevute, produce una nuova entità come output; come il task di ingresso, è un task generativo che genera una nuova entità con il suo stato iniziale:

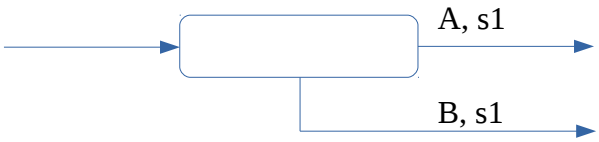


un altro tipo di pattern corrisponde ad un mapping task: riceve un'entità e il relativo stato in input e restituisce un'altra entità con il relativo stato in output; l'entità restituita è legata all'entità di input attraverso una qualche relazione tramite il quale viene effettuato il mapping (corrispondenza):

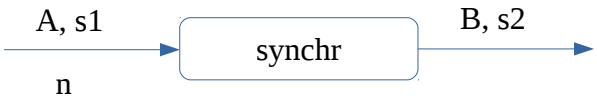
Un altro pattern corrisponde ad un matching task: riceve in input due entità diverse con i relativi stati ed emette output solamente se le due entità soddisfano una certa condizione di matching (uguaglianza), congiungendo i due flussi:



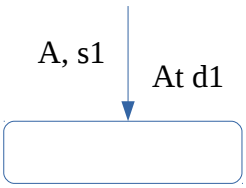
Un altro tipo di pattern corrisponde ad un forking task: riceve una sola entità in input e emette 2 entità distinte in output con i relativi stati, dividendo il flusso:



Un altro pattern corrisponde ad un task di sincronizzazione: riceve in input un numero n di entità ed emette un'entità in output solo quando sono state ricevute tutte le n entità di input:



Infine l'ultimo tra i pattern più utilizzati è quello che corrisponde ad un task temporizzato: riceve in input un'entità e dopo una certa scadenza (ad esempio "d1") esegue il task:



Grazie a questi pattern è possibile rappresentare i principali tipi di task, esistono tuttavia casi più complessi in cui bisogna utilizzare dei pattern specifici; come nel caso della scelta multipla e della scelta automatica, ma nei nostri esempi andremo ad affrontare dei casi che non necessiteranno task in cui bisogna effettuare una scelta.

Capitolo 6

L’ambiente di simulazione

6.1 Introduzione

In questa tesi abbiamo voluto approfondire la simulazione ad eventi discreti nell’ambito dei processi di business B2B; infatti la tesi estende la simulazione dal livello di building block, applicandola al caso specifico dei processi B2B. Nei primi capitoli abbiamo spiegato il funzionamento generale di una simulazione ad eventi discreti analizzandone le componenti e in seguito abbiamo esaminato i prodotti che attualmente rappresentano lo stato dell’arte. Questi si focalizzano sui processi industriali, legati agli impianti di produzione e al supporto decisionale soprattutto nei casi di decisioni critiche, ma nessuno di essi è stato sviluppato per simulare un generico processo di business B2B, ossia un processo che coinvolge più organizzazioni con sistemi informativi diversi.

Abbiamo in seguito esaminato lo standard attuale evidenziandone gli elementi e la sua applicabilità: gli elementi BPSim non erano sempre applicabili, molti di questi non erano adatti al caso dei processi di business B2B. Tuttavia, questo tipo di processi si adatta molto bene ad una rappresentazione attraverso reti di Petri, in quanto i token possono trasportare degli oggetti, passando da un’organizzazione all’altra. Per questo motivo, per scrivere il codice ed implementare l’ambiente di simulazione, abbiamo utilizzato il linguaggio Java; mentre per costruire il modello informativo, il modello collaborativo e il process model dei casi studiati, abbiamo utilizzato la notazione UML di cui abbiamo parlato nello scorso capitolo.

Poichè l’ambiente doveva essere in grado di simulare qualunque tipo di rete di Petri, esso è stato progettato per un supporto generico, fornendo le funzioni necessarie per costruire il modello da simulare e lasciando i dettagli a moduli esterni specifici in base al caso studiato. Per questo motivo, l’ambiente di simulazione per processi di business B2B sviluppato ha una funzione di libreria, dalla quale prendere le funzioni necessarie a simulare un certo modello di business. Il package in cui sono contenute tutte le classi è “simulator”; pertanto le classi avranno come nome completo “simulator.classe”, ma per semplicità le indicheremo con il nome semplice che non comprende il package di appartenenza.

6.2 La classe Net

La struttura di una rete di Petri si compone di posti, transizioni e link di input e di output; pertanto la classe Net implementa una rete e contiene tutte le informazioni riguardo le sue componenti e i tokens che sono presenti al suo interno. Queste informazioni sono contenute all'interno degli attributi della classe Net:

Attributo	Descrizione
String : name	Una stringa contenente il nome della rete. Il nome dev'essere univoco all'interno di una simulazione e viene definito in un modulo esterno alla libreria, sfruttando il costruttore della classe Net.
String : packageName	Una stringa contenente il nome del package in cui sono contenute le classi delle entità coinvolte della rete e dei task che la simulazione dovrà eseguire. Il nome dev'essere univoco all'interno di una simulazione per evitare ambiguità nella ricerca delle classi
Map<String, P> : placeMap	Una mappa per immagazzinare le informazioni riguardanti i posti della rete. La chiave di ricerca della mappa è il nome del posto e il valore è l'oggetto posto corrispondente.
Map<String, T> : transitionMap	Una mappa per immagazzinare le informazioni riguardanti le transizioni della rete. La chiave di ricerca della mappa è il nome della transizione e il valore è l'oggetto transizione corrispondente.
List<Entity> : entities	Una lista di oggetti di tipo Entity che rappresenta la lista di token contenuti in questa rete.
Int : keyGen	Un numero intero utilizzato per assegnare progressivamente un ID a tutte le Entity che vengono aggiunte nella rete.
Map<Integer, Entity> : entityByKey	Una mappa per immagazzinare le informazioni riguardanti le entità della rete. La chiave di ricerca della mappa è l'ID dell'entità e il valore è l'oggetto entità corrispondente.

Map<String, List<Entity>> : entityListByType	Una mappa per immagazzinare le informazioni riguardanti le entità della rete. La chiave di ricerca della mappa è il tipo di token e il valore è la lista di oggetti entità corrispondenti.

Ogni attributo è stato scelto pensando all'utilizzo e ai vantaggi che tale scelta avrebbe comportato nell'ambiente di simulazione: l'attributo nome è necessario per poter simulare e analizzare più reti contemporaneamente (è il caso dei processi B2B); il nome del package è fondamentale per un corretto funzionamento della simulazione, perché il simulatore non sa a priori dove trovare le classi del caso specifico che si andrà a simulare (sono in un altro package); le mappe di posti e transizioni garantiscono l'univocità dei nomi perché, essendo essi le chiavi delle mappe, non sono ammessi duplicati. I restanti attributi sono stati pensati per immagazzinare informazioni necessarie al simulatore per simulare processi B2B: l'ID univoco serve per associare un oggetto di una richiesta di un'altra organizzazione ad un oggetto specifico che viene creato con le informazioni contenute nella richiesta; la lista di entità è utile per tenere traccia delle entità presenti nella rete e le mappe restanti vengono utilizzate per velocizzare la ricerca di un oggetto Entity partento dall'ID o dal tipo di token.

La classe Net presenta anche una serie di metodi per costruire una rete e aggiornare i dati in essa contenuti:

Metodo	Descrizione
Net(String name, String package)	Il costruttore della classe Net. Riceve 2 parametri di tipo String: il primo indica il nome della rete; il secondo il nome del package con le classi contenenti le entità e i task della rete. Vengono inizializzati gli attributi corrispondenti.
Net *p*(String placeName, String typeName)	Crea un oggetto P (posto) e lo aggiunge alla rete. I parametri richiesti sono 2 String: il primo indica il nome del posto, il secondo il tipo di token ammessi; restituisce la rete stessa. Se tale posto esiste già, non apporta modifiche.
Net *t*(String transitionName, Integer delay)	Crea un oggetto T (transizione) e lo aggiunge alla rete. Richiede 2 parametri: uno String che indica il nome della transizione e un intero che indica il delay associato alla transizione e quindi il tempo

	necessario all'esecuzione del task ad essa associato; restituisce la rete stessa. Se tale transizione esiste già, non apporta modifiche.
Net tSend(String transitionName, Integer delay)	Variante del metodo precedente adatta per una transizione che prepara ed invia un messaggio ad un'altra organizzazione.
Net tReceive(String transitionName, Integer delay)	Un'altra variante del metodo *t* adatta per una transizione che riceve ed elabora un messaggio proveniente da un'altra organizzazione.
Net *l*(String link)	Crea un collegamento nella rete tra un posto e una transizione e viceversa. Riceve un unico parametro che è una stringa nel formato: "$nome_1 \rightarrow nome_2$" dalla quale ricava i nomi del posto e della transizione da collegare. Effettua un controllo sull'esistenza del posto e della transizione associata e restituisce la rete stessa. In caso di link non valido, non apporta modifiche.
Entity *addEntity*(Entity e)	Aggiunge una nuova entità alla lista entities associandogli un ID univoco. L'entità viene anche aggiunta alle due mappe entityByID e entityListByType (creando una nuova lista del tipo di token associato o aggiungendola a quella già presente). Restituisce l'entità aggiunta.
void *getFirings*()	Stampa a video l'elenco delle transizioni della rete e il numero totale di transizioni eseguite.
void *getNofTokensInPlaces*()	Stampa a video l'elenco dei post della rete e il numero di token in essi contenuti.
void *getEntities*()	Stampa a video la lista di entità della rete e il numero di token in essi contenuti.
List<Entity> *getEntitiesByType*(String type)	Sfruttando l'attributo entityListByType, restituisce la lista di entità del tipo specificato dal parametro type.
Entity *getEntityByKey*(Integer key)	Sfruttando l'attributo entityByKey, restituisce l'entità con ID uguale al parametro key.

String *getName*()	Restituisce l'attributo name.
Map<String, P> *getPlaceMap*()	Restituisce l'attributo placeMap.
Map<String, T> *getTransitionMap*()	Restituisce l'attributo transitionMap.
String *toString*()	Restituisce una stringa contenente il nome della rete.

Nell'implementazione di questi metodi, sono sorte diverse problematiche: Ad esempio nella creazione di una transizione, come associare la transizione alla porzione di codice da eseguire per tale transizione? La soluzione scelta, sfrutta il metodo statico *forName* della classe Class che riceve una stringa contenente il nome completo di package della classe della transizione ("packageName.transitionName"), esso restituisce un oggetto di tipo Class<TCode> e attraverso il metodo *newInstance* di Class otteniamo l'istanza del tipo di classe contenuto all'interno dell'oggetto Class, ossia un TCode (parleremo di questa classe nel dettaglio in questo capitolo). Infine quest'oggetto viene passato al costruttore di T che effettuerà l'associazione tra la porzione di codice specifica e l'oggetto derivante dalla classe di libreria. La stessa procedura viene utilizzata nella *tSend* e nella *tReceive*, con la differenza che, essendo due varianti del caso di transizione semplice, in esse viene definito anche il tipo di transizione T attraverso un enum: il tipo sarà send, receive o normal a seconda del metodo utilizzato per creare la transizione della rete. Questa soluzione fornisce 3 metodi diversi, a seconda della transizione che si vuole creare, ma non è l'unica soluzione possibile: un'altra soluzione consiste nell'aggiunta di un parametro nel metodo *t* che indica il tipo di transizione che si vuole creare (normal, send, receive), ma non è stata adottata per evitare di complicare la costruzione della rete aggiungendo parametri che sicuramente avrebbero ottimizzato le funzioni di libreria, ma avrebbero reso l'utilizzo dell'ambiente più difficile ad un addetto esterno il cui compito è solamente quello di scrivere un modulo per simulare la rete (un *main*). Inoltre, nella fase di sviluppo dell'ambiente di simulazione, si è partiti da esempi semplici che non utilizzavano transizioni send e receive perché la simulazione è applicabile anche a processi interni (non B2B), e tale soluzione avrebbe richiesto un parametro ulteriore che di fatto sarebbe stato sempre "normal".

Un'altra problematica riguardava il tipo di token di Entity: come ricavarlo da un oggetto di tipo Entity? La soluzione scelta ha sfruttato i metodi *getClass* e *getSimpleName*: il primo fornisce un oggetto con le informazioni sulla classe dell'entità su cui viene chiamato; il secondo restituisce una stringa contenente il nome semplice della classe, ossia al nome completo viene tolta la parte che indica il package di appartenenza. Altre soluzioni, come l'aggiunta di un parametro che ne specificasse il tipo di token o l'aggiunta di un attributo all'interno di Entity, avrebbero complicato

questa operazione di ricerca che viene effettuata in molti punti nel codice; sfruttare il nome della classe ci è sembrata la soluzione migliore.

L'ultima problematica affrontata riguarda la creazione del link: il formato "$nome_1 \rightarrow nome_2$" è molto intuitivo e permette in un solo colpo di assegnare il secondo oggetto alla lista di output del primo oggetto e il primo alla lista di input del secondo; effettuando uno *split* alla stringa (con delimitatore "→") e ottenendo dalle mappe placeMap e transitionMap gli oggetti corrispondenti ai nomi indicati. Analizzando le possibilità, si possono ottenere solamente 2 casi: se il primo oggetto è un posto, il secondo sarà una transizione e viceversa, altrimenti il link non sarà valido e non verrà creato. Infine, per ogni link valido, viene effettuato un ordinamento delle liste di input/output del posto attraverso il metodo *sortLinks* di P.

6.3 La classe P

La classe P è una classe Java che implementa un posto; essa contiene una serie di attributi che determinano le caratteristiche del posto e i token in esso presenti:

Attributo	Descrizione
String : name	Una stringa contenente il nome del posto. Il nome dev'essere univoco nella rete e viene definito in un modulo esterno alla libreria, sfruttando il metodo della classe Net *p* che crea un posto con i relativi attributi e lo aggiunge alla rete.
String : tokenType	Una stringa contenente il nome della classe di token accettabili in questo posto. Il nome della classe non è comprensivo di package, e tale classe va definita in un modulo esterno. Anche in questo caso, l'attributo viene ottenuto da uno degli argomenti del metodo *p* della libreria.
List<T> : inputLinks	Una lista di oggetti di tipo T che rappresenta la lista di transizioni dalla quale è possibile raggiungere questo posto (le transizioni di input). Questo attributo sfrutta una delle regole delle reti di Petri per cui una transizione può avere collegamenti solamente verso un posto e viceversa.
List<T> : outputLinks	Una lista di oggetti di tipo T che rappresenta la lista di transizioni raggiungibili da questo posto (le transizioni di output). Questo attributo sfrutta una delle regole delle reti di Petri per cui da un

	posto si può avere collegamenti solamente verso una transizione e viceversa.
List<Entity> : tokenList	Una lista di oggetti di tipo Entity che rappresenta la lista di token contenuti in questo posto. Le Entity ammesse sono quelle con classe uguale a quella indicata dal tokenType.

Esaminando con attenzione gli attributi, sono sorte delle problematiche: una tra queste riguarda l'univocità dei nomi assegnati ai posti per il quale è necessario un controllo. Fare questo controllo nella classe P era possibile, ma è stato scelto di fare questo controllo nella classe Net, direttamente al momento della creazione del posto, poiché creare una variabile statica (unica e condivisa da tutti gli oggetti della classe in cui è definita) in P avrebbe impedito al simulatore di funzionare con più reti di Petri simultaneamente (reti diverse possono avere posti con ugual nome) e di fatto avrebbe quindi impedito il corretto funzionamento dei processi B2B; inoltre, seguendo il modello delle reti di Petri, la rete Net è composta da posti e transizioni pertanto è il luogo più idoneo per contenere tale informazione. La seconda problematica riguarda il tokenType: possono esserci problemi se in un posto sono ammessi tipi diversi di token; questo problema viene risolto grazie all'utilizzo della classe astratta Entity e i tipi di token estendono tale classe (ne parleremo in seguito), ma un'altra possibile soluzione sarebbe stata quella di avere un attributo List<String> tokenTypes in cui mettere la lista dei tipi di token ammessi; ciò consentirebbe posti "misti" a patto che il controllo venga fatto prima di ogni inserimento nella lista. La Terza problematica riguarda le liste di input e output links: è stato scelto di sfruttare una delle regole delle reti di petri per il quale un link connette un posto ad una transizione e viceversa, non esistono collegamenti posto-posto e transizione-transizione; pertanto non è necessario avere degli oggetti inputLink e outputLink, ma possiamo rappresentare direttamente queste relazioni attraverso i punti di partenza (inputLinks) e i punti di arrivo (outputLinks) dei collegamenti. L'ultima problematica riguarda la tokenList: poiché i posti sono tutti di classe P, ma possono contenere diversi tokenType, non era possibile creare una lista di un tipo specifico; inoltre, questo tipo non è noto a priori ma dipende dalla specifica rete simulata. Per risolvere questo problema, abbiamo utilizzato la classe Entity, una classe astratta da cui discendono tutte le classi che implementano un tipo di token specifico e quindi sfruttando il polimorfismo di Java, possiamo creare una lista che può contenere tutti i token, il cui tipo discende da Entity.

Questi attributi vengono utilizzati e modificati dai metodi della classe P:

Metodo	Descrizione
P(String name, String typeName)	E' il costruttore della classe P. Esso riceve due oggetti String: il primo contenente il nome del

	posto; il secondo contenente il tipo dei token ammessi in tale posto. I 2 parametri vengono assegnati ai relativi attributi. Il metodo ritorna un riferimento all'oggetto di tipo P creato.
void *addOutputLink*(T outputTransition)	Aggiunge alla lista outputLinks la transizione outputTransition passata come parametro. Ritorna un void (non ha alcun valore di ritorno).
void *addInputLink*(T inputTransition)	Aggiunge alla lista inputLinks la transizione inputTransition passata come parametro. Ritorna un void.
void *addToken*(Entity token)	Aggiunge alla lista tokenList l'entità token passata come parametro. Il metodo non effettua alcun controllo sul tipo di token permessi all'interno del posto, quindi il controllo deve precedere la chiamata al metodo. Ritorna un void.
Entity *removeToken*()	Rimuove il primo token della lista tokenList e lo ritorna. Se la lista non ha nemmeno un token, l'operazione di rimozione fallisce e ritorna un Entity nulla, ossia null.
boolean *removeToken*(Entity ent)	Rimuove il token ent dalla tokenList e ritorna un boolean true. Se il token ent non è presente nella tokenList, l'operazione di rimozione fallisce e ritorna un boolean false.
void *sortLinks*()	Ordina le liste inputLinks e outputLinks utilizzando come comparatore un TComparator.
String *toString*()	Restituisce una stringa contenente il nome del posto, il tipo di token ammessi e il numero di token presenti nella tokenList.
String *getName*()	Restituisce l'attributo name.
String *getTokenType*()	Restituisce l'attributo tokenType.
List<T> *getInputLinks*()	Restituisce la lista inputLinks.
List<T> *getOutputLinks*()	Restituisce la lista outputLinks.
List<Entity> *getTokenList*()	Restituisce la lista tokenList.

In particolare, abbiamo 2 metodi che rimuovono un token dalla tokenList, questo perché a seconda dei casi il simulatore toglierà un token specifico dal posto oppure semplicemente il primo della lista. Invece, l'ordinamento delle liste contenenti i links è necessario per evitare che venga eseguita per prima una transizione senza guardia rispetto ad una con la guardia: questo perché laddove è presente una guardia, questa dev'essere controllata e, se la transizione viene abilitata, si passa alla creazione dell'evento; altrimenti si passa alla transizione che non ha la guardia. In sintesi, una guardia è una porzione di codice contenente una condizione che, se verificata, abilita la transizione, altrimenti la blocca; vedremo nei capitoli successivi diversi esempi di guardie.

6.4 La classe Entity

La classe Entity è una classe astratta che serve per definire un tipo di token: ogni tipo di token sarà rappresentto da una nuova classe che estenderà la classe Entity ereditandone attributi e metodi. Questa classe è fondamentale per il funzionamento dell'ambiente di simulazione; esso non conosce il tipo di token specifico che dovrà gestire, pertanto gestirà oggetti generici di tipo Entity sfruttando il polimorfismo e l'ereditarietà di Java. Essendo una classe astratta che rappresenta un token generico, contiene solamente attributi e metodi comuni a tutti i tipi di token che vengono utilizzati dal simulatore per la creazione e il recupero un token data la sua chiave (ID) e la sua rete (Net):

Attributo	Descrizione
Integer : key	ID univoco assegnato al token al momento del suo ingresso nella rete.
Net : net	Rete di appartenenza del token.
Integer : extKey	ID esterno che si riferisce ad un oggetto di un'altra delle organizzazioni coinvolte nel processo di business B2B.

Metodo	Descrizione
Integer *getKey*()	Restituisce l'attributo key.
Net *getNet*()	Restituisce l'attributo net.
Integer *getExtKey*()	Restituisce l'attributo extKey.
void *setKey*(Integer key)	Definisce l'attributo key.
void *setNet*(Net net)	Definisce l'attributo net.
void *setExtKey*(Integer extKey)	Definisce l'attributo extKey.

6.5 La classe T

La classe T implementa una transizione di una rete di Petri e in essa sono contenute tutte le informazioni relative ad una transizione e alla sua esecuzione; la tabella sottostante riporta gli attributi della classe T e la loro descrizione:

Attributo	Descrizione
String : name	Una stringa contenente il nome della transizione. Il nome dev'essere univoco nella rete e viene definito in un modulo esterno alla libreria, sfruttando il metodo della classe Net t (oppure una delle sue varianti) che crea una transizione con i relativi attributi e la aggiunge alla rete.
Int : delay	Indica il tempo necessario per eseguire questa transizione.
Int : firings	E' un contatore che indica il numero di volte in cui è stata eseguita questa transizione durante la simulazione.
List<P> : inputLinks	Una lista di oggetti di tipo P che rappresenta la lista di posti dalla quale è possibile raggiungere questa transizione (i posti di input). Questo attributo sfrutta una delle regole delle reti di Petri per cui un posto può avere collegamenti solamente verso una transizione e viceversa.
List<P> : outputLinks	Una lista di oggetti di tipo P che rappresenta la lista di posti raggiungibili da questa transizione (i posti di output). Questo attributo sfrutta una delle regole delle reti di Petri per cui da una transizione si può avere collegamenti solamente verso un posto e viceversa.
Ttype : tType	E' un Enum e indica il tipo di transizione contenuta all'interno dell'oggetto T che dipende dal metodo della classe Net utilizzato per la creazione dell'oggetto: usando tSend si otterrà una transizione di tipo "send"; usando tReceive si otterrà una transizione di tipo "receive"; infine utilizzando il metodo t si otterrà una transizione di tipo "normal" (che è anche il valore di default).
TCode : tCode	Un oggetto che indica il tipo di operazioni da eseguire durante l'esecuzione di questa transizione.

Per quanto riguarda l'univocità dei nomi, il discorso è analogo a quello fatto prima, quando abbiamo trattato l'univocità dei nomi dei posti, mentre i collegamenti di input e output, questa volta, sono delle liste di oggetti P (da una transizione si può raggiungere solamente un posto e viceversa). Il delay è necessario per fare avanzare il tempo durante la simulazione ed è rappresentato da un intero che viene assunto come intero e positivo (non vengono effettuati ulteriori controlli), mentre firings non serve per far funzionare la simulazione, ma viene utilizzato a fini statistici al termine della simulazione perché ricordiamo che il fine ultimo della simulazione è proprio quello di raccogliere delle statistiche sul quale basare le proprie scelte al fine di migliorare la produttività, l'efficienza e il costo del processo.

Non banale è stata la scelta di utilizzare gli attributi tType e tCode, tipici della transizione. L'attributo tType è un Enum che può assumere 3 valori: questa scelta semplifica la libreria ed evita di avere 3 differenti classi di transizioni che, di fatto, contengono gli stessi attributi e gli stessi metodi; infatti, non potendo conoscere a priori i dettagli del processo specifico, la classe T si limita a contenere le informazioni comuni a tutti i tipi di transizione e lascia la definizione dei dettagli all'interno dell'attributo tCode di tipo TCode, di cui, in seguito, parleremo nel dettaglio. Notare che l'oggetto tType serve solamente alle classi di libreria per implementare le funzioni dell'ambiente di simulazione, pertanto non è necessario conoscere tale classe, ma è importante utilizzare il metodo adatto a seconda della transizione che si vuole inserire nella rete.

I metodi della classe T sono metodi generici che vanno bene per ogni tipo di transizione:

Metodo	Descrizione
T(String name, Integer delay, TCode tCode)	Il costruttore della classe T. Riceve 3 parametri: il primo è di tipo String, indica il nome della transizione; il secondo è un Integer e indica il tempo necessario per eseguire la transizione; il terzo è di tipo TCode e indica le operazioni da effettuare durante l'esecuzione della transizione. Vengono inizializzati gli attributi corrispondenti.
void *addOutputLink*(P outputPlace)	Aggiunge alla lista outputLinks il posto outputPlace passato come parametro. Ritorna un void (non ha alcun valore di ritorno).
void *addInputLink*(T inputPlace)	Aggiunge alla lista inputLinks il posto inputPlace passato come parametro. Ritorna

	un void (non ha alcun valore di ritorno).
void *incFirings*()	Incrementa di un unità il contatore firings. Restituisce void.
boolean *enabled*()	Restituisce un valore booleano (vero/falso) che indica se la transizione è abilitata oppure no.
String *toString*()	Restituisce una stringa contenente il nome della transizione, il valore del delay e il valore del contatore firings.
int *getDelay*()	Restituisce l'attributo delay.
int *getFirings*()	Restituisce l'attributo firings.
List<P> *getOutputLinks*()	Restituisce l'attributo outputLinks.
List<P> *getInputLinks*()	Restituisce l'attributo inputLinks.
String *getName*()	Restituisce l'attributo name.
Ttype *getTType*()	Restituisce l'attributo tType.
TCode *getTCode*()	Restituisce l'attributo tCode
Void *setTTYpe*(TType tType)	Definisce l'attributo tType.

Alcuni metodi sono simili a quelli visti per la classe P, altri invece sono peculiari della classe T: ad esempio il metodo *enabled* è utilissimo per capire se una transizione è abilitata oppure no e velocizza la ricerca della transizione pronta a scoccare (ossia ad essere eseguita). Poichè una transizione è abilitata se in ogni posto di input è presente almeno 1 token (consideriamo il caso di reti con pesi unitari), l'implementazione del metodo va semplicemente a fare delle ricerche sulle liste: controlla la inputLinks contenente i posti di input della transizione e in ognuno di essi controlla la tokenList e verifica che sia presente almeno un token; questo si può fare semplicemente sfruttando il metodo *size* dell'oggetto List che restituisce il numero di oggetti presenti nella lista. Notare che la verifica della transizione abilitata poteva essere effettuata in altri modi, direttamente dai metodi della classe Simulator, ma tale soluzione sarebbe stata più onerosa e avrebbe aumentato la complessità dei metodi, pertanto abbiamo optato per la soluzione semplice, che sfrutta i collegamenti tra le transizioni e i posti. I restanti metodi sono dei semplici getters e setters che servono per ottenere o modificare il valore di un attributo.

6.6 La classe TCode

La classe TCode è una delle classi più importanti del nostro ambiente di simulazione: essa consente al simulatore di cambiare il suo comportamento a seconda dell'oggetto TCode contenuto all'interno della transizione T. Al suo interno troviamo una serie metodi di carattere generico e con un implementazione che non conosce i dettagli della singola transizione, restituendo dei valori precisi che possiamo considerare di default. Infatti per definire una transizione specifica bisogna creare una nuova classe che estende il comportamento di TCode: il concetto è simile a quello della classe Entity poiché il simulatore non conosce i dettagli della rete e questa classe funge da tramite tra la libreria e le classi definite dall'utente. A differenza di Entity, non è servito aggiungere attributi comuni a tutte le classi che estendevano TCode, ma era importante definire il comportamento di queste classi, attraverso la definizione di metodi, in modo tale che fossero congrui al funzionamento del simulatore e che non generassero errori inattesi. Pertanto, in base al valore restituito da ogni metodo, il simulatore prende la sua decisione e gestisce tale transizione; la tabella sottostante riporta i metodi della classe TCode:

Metodo	Descrizione
boolean *gDefined*()	Restituisce true se nella transizione associata è presente una guardia, altrimenti false, che rappresenta anche il valore di default.
boolean *aDefined*()	Restituisce true se nella transizione associata è presente un'azione, altrimenti false, che rappresenta anche il valore di default.
List<Entity> *g*(List<Entity> tokens)	Se la guardia è passiva, restituisce la lista dei tokens modificati dalla guardia. Se la guardia è attiva, restituisce null, che rappresenta anche il valore di default.
List<Entity> *a*(List<Entity> tokens)	Restituisce la lista dei tokens modificati a dalll'azione. Se l'azione non è implementata, restituisce il valore di default, ossia null.
boolean *isGenTask*()	Restituisce true se la transizione associata è un task generativo, altrimenti false, che rappresenta anche il valore di default.
boolean *isModTask*()	Restituisce true se la transizione associata è un task modificativo, altrimenti false, che

	rappresenta anche il valore di default.
String *getPayload*(Entity e)	Restituisce il contenuto del messaggio trasportato dal token. Se il metodo non è stato ridefinito, di default restituisce null.
List<String> *getRecipients*(Entity e)	Restituisce la lista dei destinatari del messaggio trasportato dal token. Se il metodo non è stato ridefinito, di default restituisce null.
Integer *getCorrelationKey*(Entity e)	Restituisce l'ID che si riferisce ad un oggetto del sistema informativo che viene utilizzato per correlare una risposta ad una richiesta precedentemente pervenuta. Se il metodo non è stato ridefinito, di default restituisce null.
Entity *genEntity*(String payload, Net net, String sender, Integer correlationKey)	Restituisce un'entità generata dalle informazioni ricevute nel messaggio. Se il metodo non è stato ridefinito, di default restituisce null.
Entity *updateEntity*(Entity e, String payload, Net net)	Restituisce l'entità modificata a seguito delle informazioni ricevute nel messaggio. Se il metodo non è stato ridefinito, di default restituisce null.

La comprensione di questi metodi è fondamentale per riuscire a comprendere la logica di funzionamento del simulatore, pertanto andremo ad esaminarli nel dettaglio.

I metodi *aDefined* e *gDefined* servono al simulatore per capire le caratteristiche del task che deve simulare e agire di conseguenza: se il task non ha la guardia, l'unico requisito necessario per lanciare il task è che la transizione sia abilitata (comportamento di default), il che è facilmente verificabile attraverso il metodo *enabled* di T; se il task ha la guardia, non basta controllare che la transizione sia abilitata, ma dev'essere soddisfatta una certa condizione tale per cui la guardia "lascia passare" la transizione (si parla di guardia passiva), altrimenti la guardia blocca la transizione (si parla di guardia attiva). Per quanto riguarda l'azione, se non è presente, il simulatore segue il comportamento di default delle reti di Petri, inserendo un token nei posti di output della transizione; se invece l'azione è presente, il simulatore, oltre ad inserire un token in ogni posto di output, dovrà eseguire l'azione associata al task. Questa soluzione semplifica enormemente il problema sulla diversa gestione delle transizioni e rende l'ambiente di simulazione flessibile e

dinamico poiché il suo comportamento varia nel tempo a seconda del tipo di transizione che deve gestire.

I metodi *g* e *a* vengono chiamati dal simulatore solamente se, rispettivamente, *aDefined* e *gDefined* restituiscono il valore true, altrimenti esso segue il comportamento di default effettuando le operazioni classiche delle reti di Petri. Poichè i dettagli dell'implementazione delle guardie e delle azioni dipendono dal caso specifico e dai tipi di token creati (ossia le classi create dall'utente che estendono Entity), la classe TCode si limita a dare delle "direttive" che le classi che implementano i task dovranno seguire; questo è possibile specificando le funzioni (e quindi il comportamento), i relativi parametri e i valori di ritorno. La lista di token, passata in entrambi i metodi, fornisce alla guardia e all'azione di avere un alto potenziale di ricerca ed elaborazione: nel caso della guardia, la lista comprende tutti i tokens di tutti i posti di input della transizione e quindi permette la verifica di condizioni complesse, che comprendono più oggetti (trasportati dai token) potenzialmente di tipo differente e con un alto numero di informazioni "controllabili" sotto forma di attributi di tali classi. Dopo aver verificato la condizione, la guardia restituisce la lista di token necessari per far scoccare la transizione ed è proprio questa lista che verrà inviata all'azione affinché elabori gli oggetti trasportati dai token della lista; anche in questo caso sono possibili moltissime operazioni su tali oggetti, spesso utili ai fini statistici come, ad esempio, l'aggiornamento di un contatore o di un attributo intero che rappresenta un codice e viene assegnato solamente al verificarsi di una certa condizione. Questa soluzione aumenta il potenziale dell'ambiente di simulazione che può quindi essere applicato in moltissimi scenari.

I metodi *isGenTask* e *isModTask* hanno un utilizzo analogo a *aDefined* e *gDefined*, ma sono specifici dei processi di business B2B: il primo restituisce true se la transizione è un task B2B, che collega due o più organizzazioni, genera una nuova entità nel sistema informativo dell'organizzazione ricevente il messaggio usando il metodo *genEntity* che dipende dalla classe che implementa il task specifico; il secondo restituisce true se la transizione è un task B2B e non genera una nuova entità nel sistema informativo dell'organizzazione ricevente il messaggio, ma va a modificare un'entità già esistente in tale sistema usando il metodo *updateEntity* (anch'esso dipende dalla classe del task). Anche in questo caso, questa soluzione consente al simulatore di decidere come comportarsi a seconda del tipo di transizione che deve gestire.

Anche i metodi *getPayload*, *getRecipients* e *getCorrelationKey* vengono chiamati dal simulatore solamente in caso il task sia di tipo B2B: il primo fornisce il payload del messaggio, ossia il contenuto che viene utilizzato per creare una nuova entità o modificarne una già esistente; il secondo fornisce la lista dei destinatari del messaggio, in modo che il simulatore invii il messaggio solamente alle organizzazioni idonee alla ricezione di quel tipo di messaggio contenente

determinate informazioni in un certo formato; l'ultimo viene utilizzato per correlare oggetti di sistemi informativi di organizzazioni diverse, senza condividere i dettagli del sistema informativo, ma solamente utilizzando un ID che si riferisce ad uno specifico oggetto, questo permette di collegare una risposta ad una precedente richiesta che conteneva le informazioni riguardo ad un particolare oggetto del sistema informativo. Essendo queste informazioni specifiche del modello collaborativo e quindi variabili a seconda dei protocolli di comunicazione stabiliti dalle organizzazioni che prendono parte al processo B2B, è compito dell'utente implementare tali metodi a seconda delle sue esigenze.

6.7 La classe TComparator

La classe TComparator implementa un comparatore per oggetti di tipo T, permettendo quindi di ordinare un insieme di oggetti T secondo una o più caratteristiche contenute in tali oggetti. Affinchè la simulazione funzioni correttamente, come spiegato in precedenza, è necessario che tutte le transizioni con una guardia abbiano una priorità rispetto a quelle senza guardia ed è proprio questa la funzione della classe TComparator. L'implementazione è molto semplice: la classe implementa l'interfaccia Comparator<T> (T è la classe su cui viene applicato il comparatore) e il confronto si basa sulla presenza/assenza della guardia nell'oggetto T attraverso il metodo *getTCode* della classe T, che restituisce il l'oggetto TCode associato alla transizione, e il metodo *gDefined* della classe TCode che restituisce true se la guardia è presente, false se è assente. Quindi l'oggetto T con la guardia precede nell'ordinamento quello senza guardia; se entrambi gli oggetti T non hanno la guardia, sono considerati equivalenti; invece non viene considersato il caso in cui entrambi gli oggetti T abbiano la guardia perchè è impossibile nelle reti di Petri tradizionali.

6.8 La classe Event

La classe Event implementa un evento: essendo la simulazione è a tempo discreto e dovendo gestire transizioni diverse con tempi di esecuzione diversi, è necessario introdurre il concetto di evento al fine di evitare sovrapposizioni e interferenze temporali durante la simulazione. Infatti, basandosi sul concetto di evento della simulazione a eventi discreti (di cui abbiamo parlato nel primo capitolo della tesi), un oggetto evento dovrà contenere le informazioni temporali relative all'evento che deve verificarsi e che verrà eseguito solamente quando sarà l'evento a priorità più alta tra quelli presenti

nella eventList. Pertanto la soluzione adottata utilizza questa classe per contenere le informazioni utili al corretto avanzamento della simulazione e non per elaborare dati; queste informazioni verranno consultate attraverso metodi di tipo getter e saranno memorizzate negli attributi della classe Event al momento della creazione dell'oggetto. Notare che non ha alcun senso memorizzare una lista degli eventi trascorsi, in quanto tutte le informazioni utili a fini statistici sono contenute negli oggetti che implementano i vari tokens (estendendo la classe Entity) e sono specifici del caso preso in esame; inoltre il simulatore stampa su video le informazioni più importanti riguardanti gli eventi simulati come il tempo di inizio, il tempo di fine e il nome della transizione.

La tabella sottostante riporta l'elenco degli attributi della classe Event:

Attributo	Descrizione
int : startTime	Contiene un numero intero positivo che indica l'istante in cui inizierà l'esecuzione dell'evento.
int : endTime	Contiene un numero intero positivo che indica l'istante in cui terminerà l'esecuzione dell'evento.
T : transition	Contiene la transizione da eseguire per simulare l'evento. Essa conterrà tutte le informazioni utili al simulatore per simulare il task associato.
List<Entity> : tokenList	Contiene la lista dei token necessari per fare scattare la transizione "transition" associata all'evento.
Net : net	Contiene la rete alla quale appartiene "transition".

Gli attributi startTime ed endTime consentono di evitare errori, da parte del simulatore, durante la sovrapposizione di due eventi: se un evento E_1 inizia all'istante $t_1=1$ e termina all'istante $t_{10}=10$, e un altro evento E_2 inizia all'istante $t_4=4$ e termina all'istante $t_8=8$, come si comporta il simulatore? In una situazione reale, dovrebbe iniziare E_1, in seguito iniziare E_2, poi terminare E_2 e infine terminare E_1; ma questo non accade nella simulazione a tempo discreto, in cui viene eseguito un evento alla volta, infatti verrà eseguito prima l'evento con l'endTime più basso, ossia E_2 e successivamente, se nel frattempo non vengono inseriti altri eventi a priorità maggiore nella eventList, E_1. Questa è una limitazione della simulazione a eventi discreti che viene risolta attraverso l'utilizzo di oggetti Event e un comparatore specifico per eventi EventComparator, di cui parleremo in seguito. Gli attributi transition e tokenList forniscono al simulatore le informazioni necessarie per simulare l'evento, mentre net è necessario quando si ha a che fare con processi che coinvolgono 2 o più reti, come nel caso dei processi B2B.

I metodi della classe sono riassunti dalla tabella sottostante:

Metodo	Descrizione
Event(T transition, Integer endTime, Integer startTime, List<Entity> tokenList, Net net)	E' il costruttore della classe Event. I 5 parametri passati servono per inizializzare i 5 attributi della classe Event.
String *toString*()	Restituisce una stringa contenente il nome della transizione, il nome della rete, il valore dell'endTime e il valore dello startTime.
Integer *getEndTime*()	Restituisce l'attributo endTime.
Integer *getStartTime*()	Restituisce l'attributo startTime.
T *getTransition*()	Restituisce l'attributo transition.
List<Entity> *getTokenList*()	Restituisce l'attributo tokenList.
Net *getNet*()	Restituisce l'attributo net.

Come detto in precedenza, si tratta solamente di metodi getters, questo implica che, una volta creato un evento, esso non possa in alcun modo essere modificato dall'ambiente di simulazione; di fatto, non occorre modificare l'oggetto evento durante la sua simulazione.

6.9 La classe EventComparator

Questa classe implementa un comparatore per oggetti di tipo Event, consentendo alla coda a priorità, ossia alla eventList, di ordinare gli eventi da quello a priorità più alta a quello a priorità più bassa. I criteri di ordinamento dovevano assicurare il corretto funzionamento dell'ambiente e quindi evitare situazioni in cui il simulatore lanciasse il task errato, come nell'esempio fatto nel precedente paragrafo. Pertanto, essendo un ambiente di simulazione a eventi discreti, l'unico criterio che avrebbe assicurato il corretto funzionamento doveva basarsi sull'endTime (istante di fine dell'evento), questo perché, una volta terminato un evento, il simulatore andrà ad analizzare il nuovo stato della rete ed eventualmente ad aggiungere nuovi eventi nella eventList. Se invece avessimo scelto lo startTime (istante di inizio dell'evento), in alcune situazioni, come quella descritta dall'esempio del paragrafo precedente, l'ordinamento nella coda avrebbe portato ad un ordine di esecuzione dei task scorretto, con conseguente sfasamento dei tempi e inquinamento delle statistiche. Pertanto il EventComparator implementa un Comparator<Event> e per decidere quale evento ha priorità più elevata chiama il metodo *getEndTime* della classe Event e confronta i due valori ricevuti.

6.10 La classe Simulator

La classe Simulator è la classe principale del nostro ambiente di simulazione per processi B2B ed implementa un oggetto simulatore. Il simulatore implementa la logica di una simulazione a eventi discreti: ha il compito di gestire l'ordine di esecuzione dei task del processo, attraverso l'utilizzo di una coda a priorità. Quando una transizione può scoccare, il simulatore crea un nuovo evento e lo inserisce nella coda e una volta esaminate tutte le transizioni, fa partire il primo evento della coda eseguendo il task ad esso associato e infine aggiorna la coda. Per far questo, la classe ha bisogno di alcune variabili memorizzate sotto forma di attributi della classe Simulator:

Attributo	Descrizione
Int : currentTime	Indica l'istante di tempo corrente della simulazione.
List<Net> : nets	Contiene la lista delle reti coinvolte nei processi da simulare.
PriorityQueue<Event> : eventList	Una coda a priorità che contiene gli eventi creati dal simulatore.

Nel dettaglio, l'attributo currentTime permette al tempo di avanzare durante la simulazione, viene utilizzato per creare gli eventi e per capire quando terminare la simulazione. Esso ha inizialmente il valore 0 e viene aggiornato ogni volta che si verifica un evento fino a che la simulazione non termina; implementando il concetto di timer delle simulazioni a eventi discreti di cui abbiamo parlato nel primo capitolo della tesi.

L'attributo nets invece è specifico dei processi B2B, in cui sono coinvolte più reti derivanti da diverse organizzazioni l'ambiente deve simulare il comportamento B2B e quindi sincronizzare le varie reti e inviare i messaggi secondo il modello collaborativo.

Infine l'eventList è la struttura dati utilizzata per gestire l'ordine di esecuzione degli eventi: essendo una coda a priorità, ossia un oggetto di tipo PriorityQueue, l'ordinamento viene effettuato in automatico ad ogni inserimento o cancellazione dalla coda; pertanto il simulatore deve gestire solamente gli inserimenti e le cancellazioni. Al momento della creazione di un oggetto Simulator, questo attributo viene inizializzato con una coda a priorità in cui viene specificato il comparatore da utilizzare per confrontare due elementi della coda, ovvero un oggetto EventComparator.

Per comprendere meglio il funzionamento del simulatore, è necessario esaminare i metodi della classe Simulator:

Metodo	Descrizione
Simulator(List<Net> nets)	E' il costruttore della classe Simulator. Riceve come parametro una lista di reti e inizializza l'attributo corrispondente.
void *putToken*(P place, Entity entity, Net net)	Inserisce nel posto place, della rete net, l'entità entity (ossia il token).
void *run*(Integer endTime)	Rappresenta il comando per far partire la simulazione. Il parametro passato rappresenta l'istante in cui la simulazione deve terminare.
void *completeEvent*(Event first)	Metodo privato della classe Simulator. Riceve un oggetto Event ed esegue il task in esso contenuto.
void *print*()	Metodo che fornisce la stampa a video dei dati principali ricavati dalla simulazione.

Il costruttore della classe Simulator riceve solamente un parametro, ossia le reti coinvolte nel process model. L'inizializzazione degli attributi endTime ed eventList non dipendono dal tipo di processo specifico, ma assumono sempre gli stessi valori iniziali e quindi non occorre specificare alcun parametro. In verità, si potrebbe evitare di specificare anche il parametro nets, ma occorrerebbe fornire attraverso un altro metodo la possibilità di aggiungere altre reti al modello, ad esempio con un metodo *addNet*, evitando l'utilizzo esplicito della lista; ma nella nostra soluzione abbiamo lasciato tale onere all'utente (addetto) finale.

Il metodo *putToken* viene utilizzato non solo per inserire i tokens iniziali nella rete, ma anche per gli inserimenti successivi, infatti il passaggio di un token da un posto di input ad un posto di output avviene in due fasi: prima vengono tolti i tokens dai posti di input e viene creato ed inserito nella coda un nuovo evento; in seguito verrà eseguito il task associato all'evento e inseriti i tokens nei posti di output. Il metodo, essendo fondamentale per configurare il modello della simulazione, effettua dei controlli sull'esistenza del posto in cui verrà inserito il token e se tale posto può accettare token di quel tipo; in caso contrario verrà generata un eccezione di classe Exception (manca il controllo sulla rete, assunta corretta). L'inserimento di un token in un posto della rete viene fatto sfruttando il metodo *addToken* della classe P, ma va fatta una distinzione tra i due casi possibili: il token viene inserito nella rete per la prima volta oppure era già presente nella rete ed è stato spostato da un posto ad un altro. Nel primo caso, si dovrà aggiornare la lista entities di Net chiamando il metodo *addEntity*, mentre nel secondo caso questo passaggio va ignorato. Questo è

realizzabile attraverso una condizione *if* che va a controllare se l'attributo key dell'entità trasmessa, ossia l'ID del token, è pari a -1 (valore di inizializzazione) oppure no: se la condizione è vera, si tratta di una nuova entità, altrimenti di un'entità già presente nella rete indicata. Il passaggio successivo segue le regole delle reti di Petri e consiste nel fare un loop che fa ad esaminare tutte le transizioni di output del posto passato come parametro:

- Per ciascuna transizione si va a vedere se è abilitata sfruttando il metodo *enabled* della classe T; se non lo è si passa alla transizione successiva.
- Se è abilitata si verifica la presenza di una guardia usando il metodo gDefined della classe TCode.; se la guardia è non presente, la transizione può scoccare. Viene quindi creato un nuovo evento (chiamando il costruttore della classe Event) con tale transizione, startTime pari al currentTime della simulazione e l'endTime pari allo startTime+delay, la rete net e la lista dei tokens necessari per fare scoccare la transizione (nel nostro caso 1 token da ogni posto di input) che vengono rimossi dai posti in cui erano contenuti.
- Se la guardia è presente, bisogna verificare la condizione imposta dalla guardia. Questo viene fatto usando il metodo *g* della classe TCode, a cui vengono passati tutti i tokens di tutti i posti di input per quella transizione: se la condizione è vera, la guardia restituisce la lista dei tokens necessari per fare scoccare la transizione e viene creato un nuovo evento sfruttando il costruttore come nel punto precedente; in caso contrario, la guardia blocca la transizione e il simulatore passerà ad esaminare la transizione successiva, se presente.

Il metodo *run* è il motore del simulatore ed è composto da un ciclo infinito che effettua le seguenti operazioni:

- Attraverso il metodo *peek* della classe PriorityQueue, che restituisce il primo elemento della coda a priorità senza rimuoverlo, viene ottenuto il tempo di fine del prossimo evento e si assegna tale valore al currentTime, usando il metodo getEndTime della classe Event.
- Se il valore supera l'endTime di fine simulazione, si esce dal ciclo e la simulazione termina, altrimenti si prosegue recuperando e rimuovendo il primo elemento della coda, usando il metodo *poll* della classe PriorityQueue.
- Infine si chiama il metodo completeEvent della classe Simulator, passandogli l'evento appena recuperato dalla coda.

Quindi dalla simulazione si esce solamente in 2 casi: il currentTime raggiunge l'endTime e quindi la simulazione termina; oppure la eventList non contiene nessun evento e quindi il ciclo si blocca, sintomo che la rete presenta un deadlock.

Il metodo completeEvent ha una complessità variabile che dipende dal tipo di transizione associata all'evento in corso di completamento:

- Se la transizione ha TType uguale a normal, siamo nel caso semplice: il simulatore controlla se è presente un'azione oppure no; se presente, viene eseguita chiamando il metodo *a* della classe TCode alla quale viene passata la lista di token dell'evento, ottenuta chiamando il metodo getTokenList della classe Event. In seguito vengono inseriti i token nei posti di output della transizione usando il metodo putToken descritto in precedenza ed effettuando un controllo sul tipo di token ammesso per evitare di mettere token errati nei posti errati. Questo controllo viene fatto usando il metodo *getClass* di Object che restituisce la classe dell'oggetto su cui viene applicato (in questo caso il token che estende Entity) e su di esso viene chiamato il metodo *getSimpleName* che rimuove dal nome la parte contenente il package in cui è contenuta la classe.
- Se invece la transizione ha TType uguale a send, la procedura è più complessa perché siamo nel caso di una transizione specifica per processi B2B che deve inviare un messaggio ad un'altra organizzazione. Per ogni token della tokenList dell'evento, viene applicata una procedura, divisa in due fasi, che permette di far passare questo token da una rete all'altra: la prima fase è quella di spedizione in cui attraverso il metodo *getPayload* della classe TCode si ottiene il corpo del messaggio da inviare; attraverso *getRecipients* si ottengono i nomi delle reti a cui inviare il messaggio; attraverso il metodo *getCorrelationKey* si ottiene la chiave necessaria per correlare l'oggetto inviato nel messaggio. A questo punto la fase di spedizione è stata simulata, inizia la fase di ricezione: per ogni destinatario il simulatore va a cercare la transizione omonima (l'omonimia è necessaria per effettuare il collegamento) nella sua rete (il simulatore ha la lista di reti nets e quindi basta confrontare il nome del destinatario con quelli delle reti) e verifica che essa sia una transazione con TType uguale a receive. In tal caso, il task TCode associato alla transizione potrà essere di 2 tipi: generativo o modificativo; informazione ottenibile usando i metodi *isGenTask* e *isModTask* della classe TCode. Nel primo caso, verrà chiamato il metodo *genEntity* del task associato alla transizione, che genererà una nuova entità nel sistema informativo del destinatario che rappresenta il token "passato" dalla rete precedente che verrà inserito nel posto di output della transizione usando il metodo *putToken*. Nel secondo caso, verrà invece chiamato il metodo *updateEntity* modificando un'entità già presente nel sistema informativo del destinatario che verrà inserita nel posto di output della transizione. Termina quindi anche la fase di ricezione e il token circolerà nella "nuova" rete.

Durante queste fasi, vengono anche aggiornati i contatori firings delle transizioni e infine il metodo ritorna il controllo al chiamante che aggiornerà il currentTime e completerà il prossimo evento.

L'ultimo metodo della classe Simulator è *print* che è utile per trarre delle statistche e studiare possibili miglioramenti del sistema o risolverne le problematiche: per ogni rete, vengono chiamati i metodi *getFirings*, *getNofTokensInPlaces* e *getEntities* della classe Net; in seguito vengono stampate le informazioni che riguardano i token che sono rimasti "nel mezzo", cioè quelli che sono stati tolti dai posti di input, ma non sono ancora stati inseriti nei posti di output e quindi non sono presenti in nessuna rete. E' il caso dei tokens contenuti negli oggetti Event, presenti nella eventList a cui solo il simulatore può accedere (non esistono metodi getters); pertanto vengono stampate le informazioni relative a questi tokens che vengono definiti token "pendenti".

Nei capitoli successivi andremo ad esaminare vari esempi di applicazione dell'ambiente di simulazione proposto, partendo da un esempio semplice e arrivando ad un esempio B2B completo.

Capitolo 7

Esempio semplice

7.1 Production System 1

Nello sviluppo dell'ambiente di simulazione, siamo partiti da un esempio semplice chiamato Production System 1che rappresenta un sistema di produzione composta da una macchina e un carrello. La macchina produce una componente di un prodotto in 8 unità di tempo, poi la mette su un carrello e l'operazione dura 1 unità di tempo, in seguito fa una pausa di un 1 unità di tempo e poi riprende producendo una nuova componente. Il carrello può contenere solamente una componente alla volta, quindi, ricevuta la componente dalla macchina, esso dev'essere sostituito da un carrello vuoto in grado di ricevere la componente successiva, questa operazione avviene in 5 unità di tempo. L'unità di tempo può rappresentare qualunque metrica utilizzata per misurare l'avanzamento del tempo: secondi, ore, minuti, giorni, mesi, anni e così via; nel nostro caso, potremmo immaginare di avere a che fare con dei minuti, ma per semplicità lasceremo l'unità di tempo generica UT. Questo è un esempio semplice di Time Marked Graph, ossia di grafo marcato temporizzato, poiché alle transizioni (tasks) è associato un delay.

Per poter simulare questo caso, è necessario definire il modello informativo e il process model, individuando le reti e le entità coinvolte nel processo di produzione, partendo da posti, transizioni, links e tokens; in seguito aggiungerci eventuali guardie e azioni. Invece, in questo caso, non è necessario definire il modello collaborativo, perché questo processo non è un processo di business B2B, infatti non ci sono altre organizzazioni coinvolte.

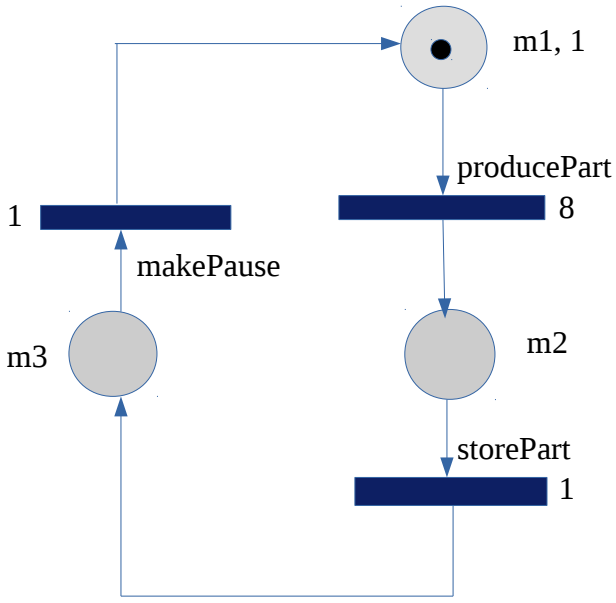
7.1.1 Il process model

Per definire il process model bisogna analizzare nel dettaglio le informazioni relative all'esempio in questione per identificare i posti e i tokens e i task del sistema. Possiamo notare che l'attività del carrello è subordinata a quella della macchina, infatti, se non vengono prodotte le componenti, il carrello rimane in attesa; pertanto ci sarà una transizione in comune che "sblocca" il carrello.

Quindi, per poter svolgere le loro attività, la macchina e il carrello avranno bisogno di almeno un token ciascuno per permettere al processo di avanzare da un task all'altro.

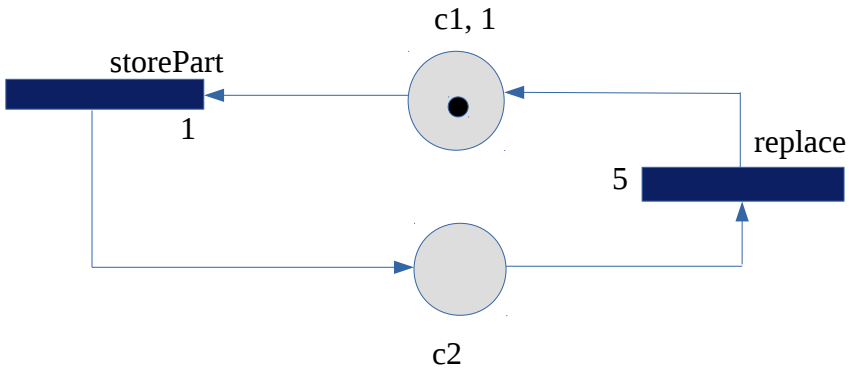
Possiamo quindi stabilire due posizioni iniziali in cui saranno presenti i 2 tokens della rete: il primo token sarà di tipo Machine (cioè macchina) e si troverà in un posto che chiamiamo m1, il secondo token sarà di tipo Container (cioè carrello) e si troverà in un posto che chiamiamo c1; si parte quindi da una situazione iniziale con macchina e carrello fermi. Il passo successivo è quello di seguire le attività di ciascuna entità e definire posti, transizioni e links.

La macchina produce una componente in 8 UT, quindi possiamo definire una transizione di tipo ProducePart con un delay pari ad 8 e collegarla ad m1 attraverso un link (una freccia) di input (per il posto sarà, viceversa, un link di output). Poichè nelle reti di Petri, un posto può essere collegato solamente ad una transizione e viceversa, storePart sarà collegata in uscita ad un posto m2, in cui depositerà il token. In seguito, la macchina deposita la componente nel carrello in 1 UT, quindi possiamo definire una transizione di tipo StorePart con un delay pari ad 1 e collegarla ad m2 attraverso un link di input. Quindi, la storePart metterà il token di tipo Machine in un posto di output m3 ad essa collegato attraverso un link di output. Infine l'ultima attività svolta dalla macchina consiste nell'effettuare una pausa di 1 UT, quindi possiamo definire una transizione di tipo MakePause con un delay pari ad 1 e collegarla al posto m3 attraverso un link di input. Infine il la sequenza di attività riparte dalla producePart, quindi chiudiamo il circuito con un link di output uscente da MakePause e entrante in m1.

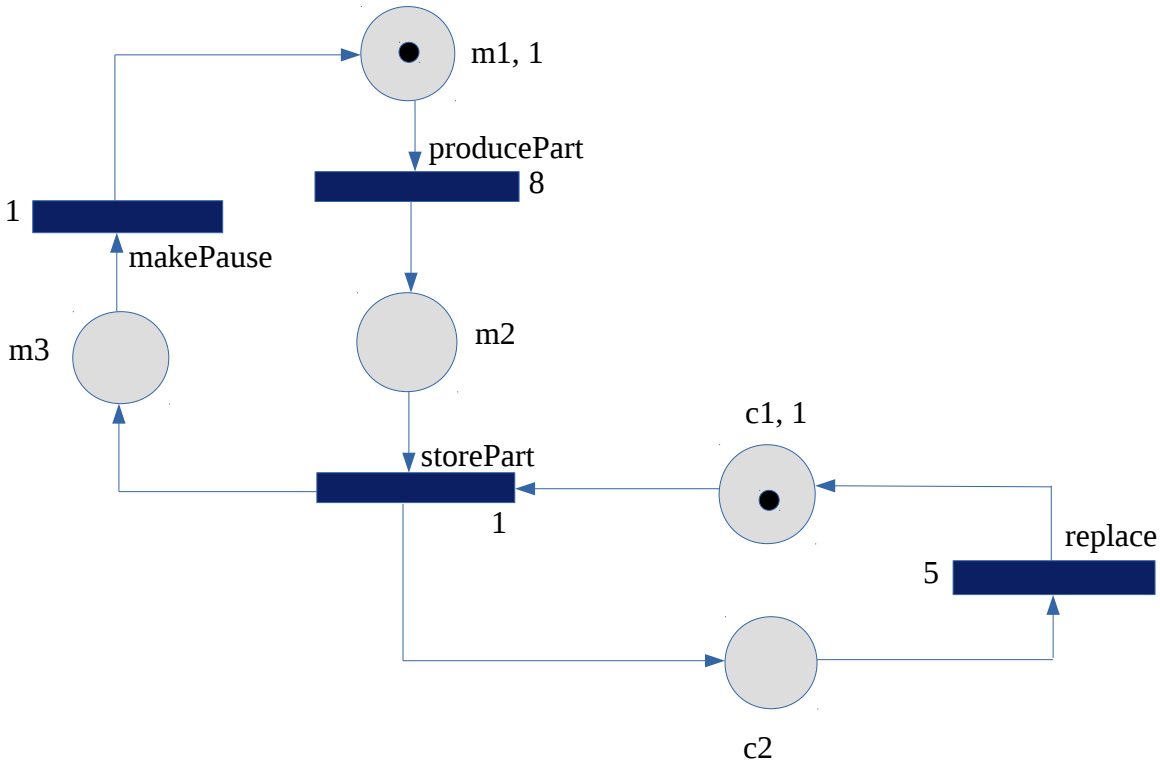


Il carrello invece resta in attesa nel posto c1 finché non viene eseguita la storePart, quindi possiamo collegarla con un link uscente dal posto. La storePart, oltre a mettere un token di tipo Machine in m3, prenderà il token contenuto in c1 e lo metterà in un posto c2 ad essa collegato tramite un link di

output. In seguito il carrello pieno verrà sostituito da un carrello vuoto in 5 UT, ma per semplicità possiamo pensare di avere un carrello solo che viene svuotato e rimesso in attesa da una transizione di tipo Replace, collegata a c2 attraverso un link di input. In seguito, il carrello torna in attesa, quindi la sequenza di attività ricomincia: inseriamo un link tra replace e c1 che chiude il circuito.

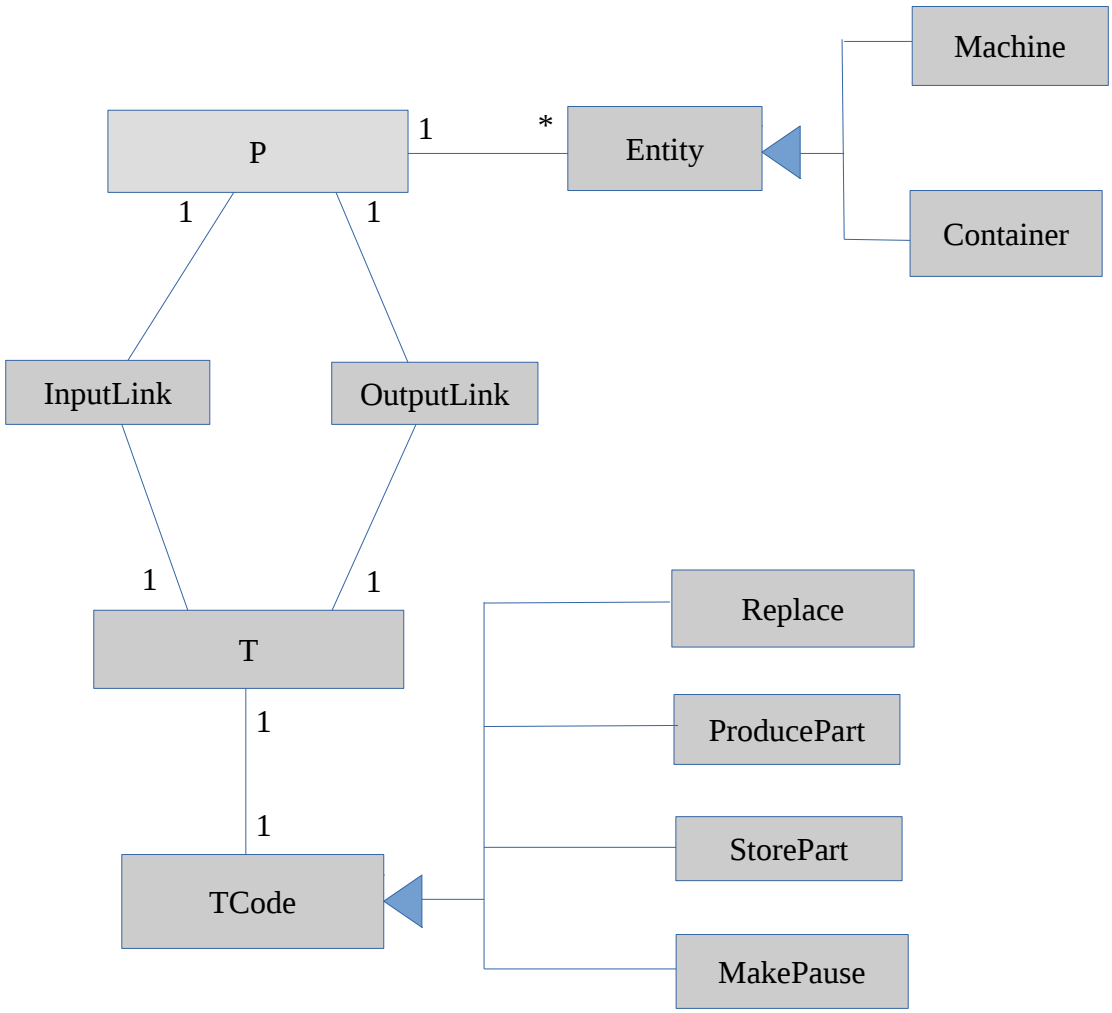


Quindi, il process model risultante sarà il seguente:



7.1.2 Il modello informativo

Il modello informativo mostra gli oggetti presenti nel sistema e le relazioni tra di essi: il modello deve quindi contenere tokens, posti, transizioni, links di input e di output. Nel capitolo precedente, abbiamo già definito delle classi Java che rappresentavano gli elementi delle reti di Petri, pertanto dobbiamo solamente scegliere come gestire gli elementi specifici del caso in questione, ossia i tokens che estenderanno la classe di libreria Entity e implementeranno i tokens specifici del sistema. Nel paragrafo precedente abbiamo stabilito l'esistenza di 2 entità diverse (Machine e Container) e di 4 task (producePart, storePart, makePause e replace) . Pertanto, sarà necessario definire classi omonime che estendono, rispettivamente, Entity e TCode; e che contengono tutte le informazioni che ci servono per verificare eventuali condizioni (in questo caso non ci sono ne guardie ne azioni) e per ricavare delle statistiche sul sistema. Quindi il modello informativo di questo esempio è molto semplice, poiché aggiunge le 2 classi di token e le 4 classi di task al modello informativo della libreria:



7.1.3 Applicazione dell'ambiente

Una volta terminata la costruzione dei modelli che descrivono il sistema, possiamo applicare l'ambiente di simulazione al modello risultante: per fare questo, è necessario scrivere una classe Main che definisce la rete e inserisce i tokens sfruttando la libreria; è necessario anche definire le classi relative ai tipi di tokens e ai tasks estendendo il comportamento delle classi Entity e TCode.

Per costruire la rete, bisogna utilizzare i metodi *p*,*t* e *l* della classe Net che inseriscono, rispettivamente, un posto, transizione e un link nella rete, restituendo un riferimento alla rete stessa. Il valore di ritorno è stato pensato per permettere all'utente di costruire la rete in modo rapido e compatto utilizzando Java-Fluent: una catena di chiamate a funzioni formata chiamando il metodo sull'oggetto restituito dal metodo precedente, di seguito riportiamo il codice:

```
Net net = new Net("Production1", "production");

net.p("m1", "Machine").p("m2", "Machine").p("m3", "Machine")
      .p("c1","Container").p("c2","Container")
      .t("ProducePart",8).t("StorePart",1).t("MakePause",1)
      .t("Replace",5).l("m1->ProducePart").l("ProducePart->m2")
      .l("m2->StorePart").l("StorePart->m3").l("m3->MakePause")
      .l("MakePause->m1").l("StorePart->c2").l("c1->StorePart")
      .l("c2->Replace").l("Replace->c1");
```

L'alternativa sarebbe stata di elencare tutte le chiamate, ma con Java-Fluent il codice risulta più chiaro e conciso.

In seguito l'utente deve definire le classi relative ai tokens: Machine e Container. Esse estenderanno la classe Entity e i loro attributi e metodi dipenderanno dalle informazioni che l'utente vuole ricavare dalla simulazione. In questo primo esempio non sono presenti né guardie né azioni, quindi non abbiamo dettagliato le classi inserendo attributi utili ai fini statistici e i metodi che gestiscono il loro valore, quindi le 2 classi sono vuote. Discorso analogo per quanto riguarda le 4 classi di task: esse estenderanno TCode, ma come per i tokens, saranno classi vuote che non modificheranno gli oggetti token.

7.1.4 Risultati Ottenuti

Questo esempio ci è servito per testare il simulatore e verificare che le tempistiche e le informazioni ottenute dalla stampa a video, attraverso il metodo *print* di Simulator, fossero corrette; inoltre, è stato possibile ottimizzare le classi di libreria attraverso l'analisi dei risultati ottenuti. Da questi si è potuto ricavare il tempo di ciclo delle transizioni, ossia l'intervallo di tempo tra due scatti consecutivi di una transizione che si può determinare solamente al termine della fase di bootstrap della simulazione (il transitorio iniziale); per raggiungerla abbiamo messo un endTime relativamente grande in modo da far durare la simulazione abbastanza a lungo. Il tempo di ciclo è risultato pari a 10 UT, che è esattamente il tempo di percorrenza del circuito della macchina, mentre quello del carrello è pari a 6 UT: questo è un dato molto importante, perché dimostra che il carrello passa 4 UT in attesa che arrivi una nuova componente, quindi abbiamo uno spreco di risorse (il carrello). Pertanto abbiamo provato nuovamente a simulare il sistema, ma stavolta aggiungendo un token in più su m1 che corrisponderebbe, nel sistema reale, all'aggiunta di una macchina: il risultato è stato una diminuzione del tempo di ciclo da 10 UT a 6 UT perché, terminato il transitorio, le transazione del circuito della macchina si ripetono ogni 6 UT e ogni token Machine percorre il circuito in 12 UT, mentre prima impiegava 10 UT il che implica che ogni token attende 2 UT. Il carrello, come nel caso precedente, resta a 6 UT e questo implica che l'attività del carrello non è più subordinata a quella della macchina, ma viceversa sono le due macchine che attendono che il carrello si svuoti per metterci la nuova componente. In conclusione, la produttività del processo aumenterebbe (quasi del doppio) con l'aggiunta di una macchina e quindi gli analizzatori di business potrebbero utilizzare queste informazioni per decidere se acquistare una macchina per aumentare la produttività del sistema o oppure no, ad esempio quando l'aumento di produttività non è richiesto perché si è già raggiunta la saturazione della domanda del prodotto.

Nel prossimo capitolo vedremo un esempio avanzato in cui questa rete viene modificata e vengono aggiunte guardie e azioni e un altro esempio con un'altra rete di Petri avanzata.

Capitolo 8

Esempi con rete di Petri avanzata

8.1 Production Business Process 2

Il secondo esempio di applicazione dell'ambiente di simulazione è un processo di business completo: il "Production Business Process 2". Esso rappresenta un sistema di produzione più complesso e più realistico rispetto all'esempio precedente che invece rappresentava una versione semplificata di un sistema di produzione reale (può essere visto come un'estensione dell'esempio precedente). In questo caso abbiamo una macchina che produce continuamente componenti: impiega 10 UT per produrne una, poi mette la componente prodotta nel carrello in 1 UT ed effettua una pausa di 2 UT ogni 10 unità prodotte. Il carrello, questa volta, può contenere fino a 40 componenti e una volta pieno, viene svuotato in 5 UT. Possiamo subito notare la presenza di due condizioni: una nel circuito della macchina e una in quello del carrello che si tradurranno in due guardie. Questo sistema può essere rappresentato da una rete di Petri avanzata in cui i token trasportano oggetti con delle informazioni che vengono modificate dalle transizioni (azioni) e che possono condizionare la scelta delle transizioni da far scoccare (guardie).

Il sistema rappresenta un processo di business, ma non è un processo di business B2B, pertanto non dobbiamo definire il modello collaborativo, ma ci limiteremo a definire il process model e il modello informativo.

8.1.1 Process model

Esaminando con attenzione le informazioni disponibili, possiamo notare delle somiglianze con l'esempio del capitolo precedente: i tipi di token sono gli stessi (Machine e Container) e tutte le transizioni dell'esempio "Production System 1" sono presenti in questo nuovo esempio (producePart, storePart, makePause e replace), ma sia i tokens che le transizioni necessitano delle modifiche per rendere il modello coerente con il nuovo sistema.

Per costruire la rete, partiamo da una situazione iniziale in cui abbiamo 1 token di tipo Machine in m1 e un token di tipo Container in c1 (m1 e c1 sono gli stessi dell'esempio precedente) e

individuiamo la posizione delle guardie e delle azioni: la prima guardia, ogni 10 componenti prodotte, ferma la macchina facendo scoccare la transizione makePause (la guardia viene quindi messa su tale transizione); invece se la condizione non è soddisfatta, continua la sua produzione. La seconda guardia, ogni 40 componenti ricevute, svuota il carrello facendo scoccare la transizione replace; se tale condizione non è soddisfatta, allora tornerà in attesa di ricevere una nuova componente. La presenza di queste due guardie implica l'esistenza di una "strada alternativa" nel caso in cui le transizioni fossero bloccate dalle guardie: pertanto nella rete verranno inserite due transizioni, ti tipo Continue1 e Continue2, parallele rispettivamente a makePause e replace che permetteranno ai tokens di avanzare nei circuiti quando la guardia è attiva sulla transizione. Queste 2 scelte sono due Free Choice perché ognuno dei due posti, m3 e c2, ha due transizioni di output per il quale è l'unico posto di input.

Invece, per quanto riguarda le azioni, esse andranno a modificare degli attributi degli oggetti trasportati dai tokens ed esaminando le condizioni imposte dalle guardie, andremo ad inserire 2 azioni nella rete, la prima nella transizione producePart che aggiornerà un contatore presente dentro il token di tipo Machine; la seconda nella transizione storePart che aggiornerà un contatore presente dentro il token di tipo Container. Notare che, era anche possibile inserire due azioni in makePause e replace che azzerano i 2 contatori, ma poiché entrambe le guardie sono passive ogni "n" (con n numero di componenti prodotte dalla macchina o ricevute dal carrello), possiamo evitare di resettare il contatore semplicemente controllando che il resto della divisione intera, tra contatore e n, dia come risultato zero:

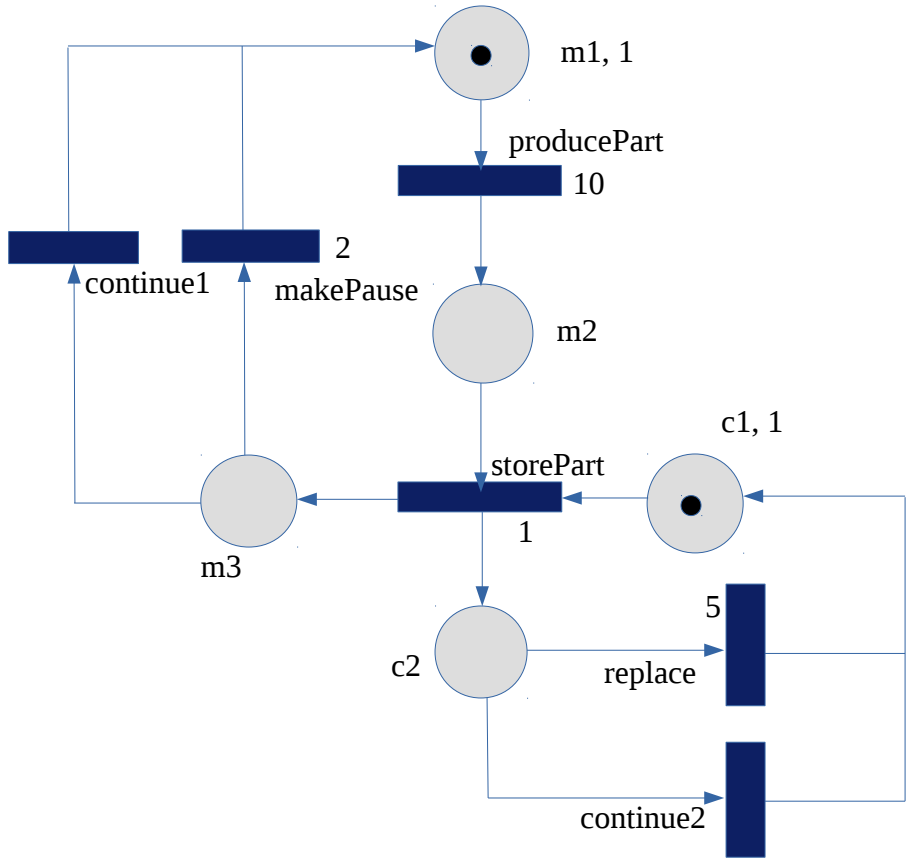
$$\text{contatore} \ \% \ n = 0$$

Quindi la rete risultante sarà una FC, composta da due sotto reti di tipo State Machine (la macchina e il carrello): partendo dalla rete dell'esempio precedente, aggiungiamo una transizione di output al posto m3, la collegata alla transizione continue1 che a sua volta sarà avrò un link di output verso m1 per chiudere il circuito; la stessa cosa viene fatta su c2, collegando il posto alla transizione continue2 attraverso un link di output e in seguito collegando la transizione a c1 attraverso un altro link di output della transizione.

Questo esempio dimostra perché è stato necessario ordinare le transizioni nei links di input e di output dei posti, infatti, se non avessimo ordinato le transizioni dando a priorità a quelle con una guardia rispetto a quelle senza guardia, le transizioni sarebbero state selezionate in base alla loro posizione nella lista che, in assenza di ordinamento, sarebbe stata causale. Se ad esempio la macchina avesse già prodotto 10 componenti e il token si trovasse su m3, se venisse esaminata prima la transizione continue1 rispetto a makePause, scoccherebbe continue1 e non scoccherebbe makePause poiché il token viene prelevato da continue1 che poi lo inserisce in m1; quindi ci

sarebbe un errore. Invece con l'ordinamento fornito attraverso il comparatore TComparator, viene prima esaminata makePause e che, essendo n pari a 10, può scoccare e quindi preleva il token e lo inserisce in m1e di conseguenza continue1, non avendo token nel posto di input, non scoccherà. Notare che nemmeno il caso in cui entrambe le 2 transizioni siano abilitate è corretto: esso porterebbe alla generazione di 2 tokens in m1, quindi la transizione che viene abilitata per prima deve crea un evento, lo inserisce nella eventList e rimuove i tokens necessari dai posti di input, in modo che le altre transizioni non possano scoccare e generare errori.

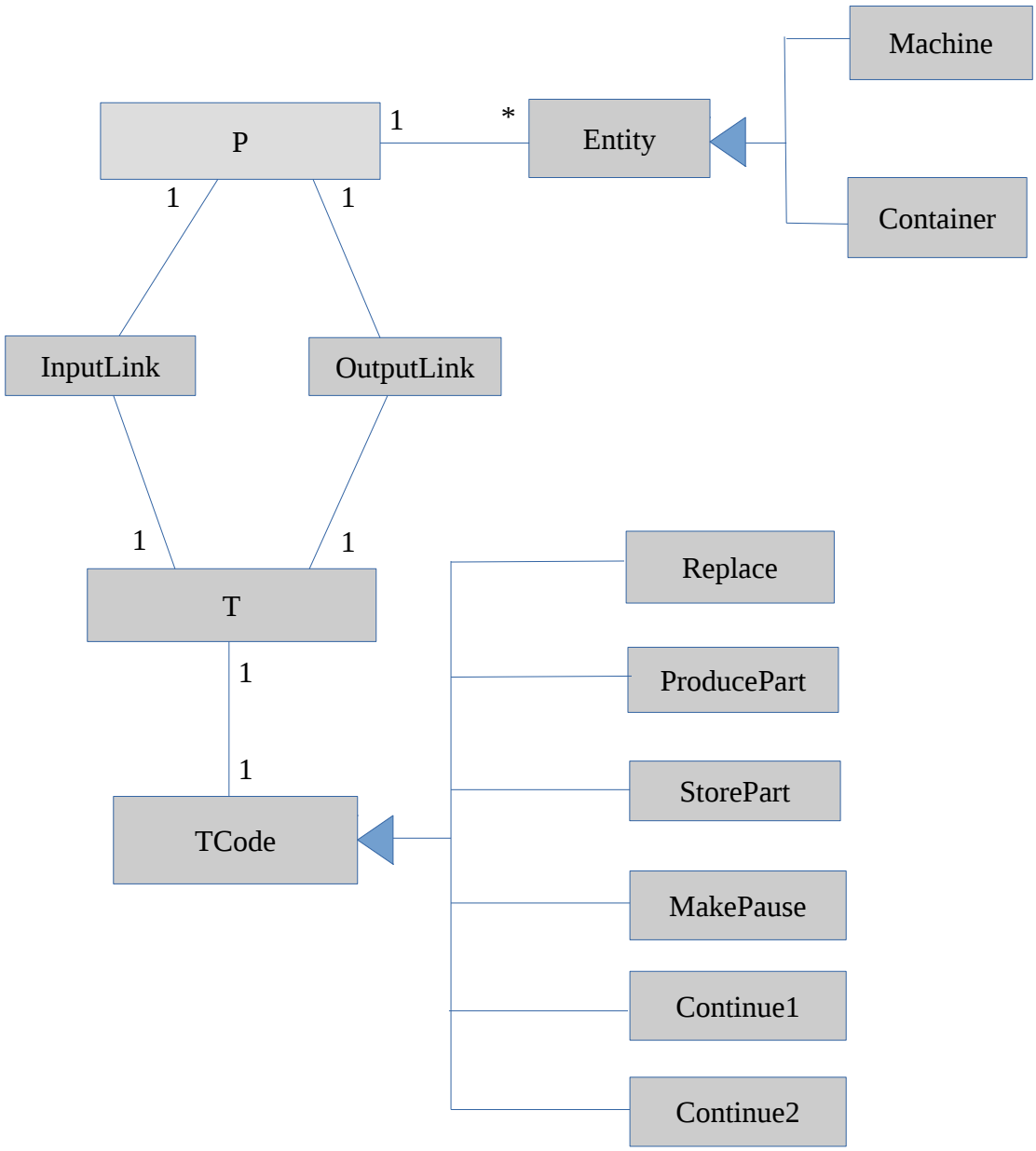
In conclusione, la rete risultante è la seguente:



Infine, analizzando la rete, possiamo dire che è live e safe.

8.1.2 Il modello informativo

Il modello informativo è molto simile a quello dell'esempio precedente, infatti presenta solamente l'aggiunta delle transizione continue1 e continue2 come estensioni di TCode:



8.1.3 Applicazione dell'ambiente

Anche in questo caso è necessario scrivere una classe Main per costruire la rete ed inserirci i tokens. La costruzione della rete è molto simile a quella dell'esercizio precedente, presenta infatti solamente l'aggiunta delle 2 nuove transizioni (continue1 e continue2) e la modifica di alcuni delay:

```
Net net = new Net("Production1", "production");

      net.p("m1", "Machine").p("m2", "Machine").p("m3", "Machine")
            .p("c1", "Container").p("c2", "Container")
            .t("ProducePart",10).t("StorePart",1).t("Continue1",0)
            .t("MakePause",2).t("Continue2",0).t("Replace",5)
            .l("m1->ProducePart").l("ProducePart->m2").l("m2->StorePart")
            .l("StorePart->m3").l("m3->Continue1").l("Continue1->m1")
            .l("m3->MakePause").l("MakePause->m1").l("StorePart->c2")
            .l("c2->Continue2").l("Continue2->c1").l("c1->StorePart")
            .l("c2->Replace").l("Replace->c1");
```

Alle transizioni continue viene associato un valore del delay pari a 0, perché di fatto, permettono alla macchina e al carrello di saltare una transizione e quindi non occupano tempo. Un'altra cosa che possiamo notare è la mancanza delle guardie durante la costruzione della rete: esse vanno definite nelle classi che implementano le transizioni estendendo il comportamento di TCode.

In seguite viene creato un oggetto Simulator passandogli la rete appena creata: su tale oggetto verrà chiamato il metodo putToken per inserire un token (un oggetto di una classe che estende Entity) in un certo posto della rete:

```
      Simulator sim = new Simulator(nets);
      try {
            for(i=0; i<machines; i++) {
                  sim.putToken("m1",new Machine(), net);
            }
            for(i=0; i<containers; i++) {
                  sim.putToken("c1",new Container(), net);
            }
      } catch (Exception e) {
            e.printStackTrace();
      }
```

In questo codice di esempio, vediamo come è possibile inserire più tokens nella rete, dove machines e containers rappresentano il numero di macchine e di carrelli a disposizione; mentre nets rappresenta la lista di reti coinvolte nel processo (in questo caso la lista ha un solo elemento). Per far partire la simulazione utilizziamo il metodo *run* di Simulator, per avere una stampa a video, chiamiamo anche il metodo *print* della stessa classe.

L’utente deve scrivere, oltre alla classe Main, anche le altre classi specifiche del processo di business: Machine e Container (i 2 tipi di token); producePart, storePart, makePause, continue1, replace e continue2 (i 6 tipi di task). Le classi Machine e Container avranno entrambe un contatore, implementato sotto forma di attributo privato che verrà consultato dalle guardie e modificato dalle azioni: serviranno quindi metodi getter per la consultazione e metodi setter per la modifica, in questo caso, poiché si parla di contatori, i metodi setter possono essere sostituiti con metodi che incrementano di un unità il valore attuale del contatore. Va inoltre ridefinito il metodo *toString* ereditato dalla classe Object affinché la stampa dell’entità comprenda l’attributo creato.
Per quanto riguarda i task bisogna implementare le guardie e le azioni scrivendo le classi ad essi associate: le transizioni continue1 e continue2 non hanno né guardia né azione, pertanto la loro implementazioni risulterà in una classe vuota che estende TCode ereditandone il comportamento. La classe producePart ha un’azione che incrementa il contatore del token di tipo Machine chiamando il metodo appositamente creato (ad esempio *incN*); quindi vanno ridefiniti e metodi *aDefined,* che deve restituire true (e non false, come in TCode) e *a*, che deve restituire la lista dei tokens eventualmente modificati dall’azione e quindi selezionare tra i tokens passati dal simulatore quelli da modificare (di tipo Machine) e aggiornarne il contatore; infine restituire i tokens della lista. In maniera analoga, anche in storePart verranno ridefiniti gli stessi metodi, con la differenza che in tale transizione passano sia token di tipo Machine che Container, quindi bisogna selezionare il token corretto, ad esempio usando il costrutto *instanceOf* per capire di che tipo di token si tratta e incrementare solamente il contatore dei token di tipo Container, mentre i token di tipo Machine dovranno comunque essere aggiunti alla lista (senza essere modificati) per poter essere inseriti nei posti di output. Questa transizione rappresenta una delle criticità incontrate poiché rappresenta l’unico punto d’incontro nella rete dei 2 circuiti. La classe MakePause dovrà ridefinire invece i metodi gDefined (che deve restituire true) e g che restituisce la lista dei tokens scelti dalla guardia se la condizione è soddisfatta, altrimenti ritorna null. Discorso analogo per la classe Replace, che dovrà ridefinire gli stessi metodi: in entrambi i casi si può sfruttare l’operatore % per la condizione della guardia e, in questo caso, non c’è ambiguità sul tipo di token poiché si trovano in due circuiti diversi della rete.

8.1.4 Risultati ottenuti

In questo esempio abbiamo utilizzato 1 macchina e un carrello, ma la simulazione può essere fatta con un numero di macchine e carrelli arbitrario: è l’utente che scrive la classe Main e quindi decide

il numero di token nella rete. Attraverso la simulazione è quindi possibile testare diverse configurazioni, analizzarne i risultati e verificare se essi coincidono con i risultati attesi.

Simulando il sistema con 1 macchina e 1 carrello e ponendo un endTime pari a 1000 UT (l'endTime grande serve per passare la fase di transitorio ed avere risultati più precisi) scopriamo che, a fine simulazione, sono state prodotte (e ricevute) 89 componenti e sono stati lanciati 356 task; nella eventList è rimasto un solo task cioè producePart perchè la simulazione è terminata mentre l'evento associato a tale task si trovava nella eventList (è un evento pendente). Il throughput è quindi di 0,089 componenti per unità di tempo. È possibile, a parità di UT, migliorare il throughput? Sono presenti bottleneck (colli di bottiglia) e se si come eliminarli? Per poter rispondere a queste domande, possiamo sfruttare l'ambiente di simulazione: un primo indizio su dove possa trovarsi il collo di bottiglia, ce l'ho fornisce la eventList in cui c'è un task del circuito della macchina che è proprio il task con delay più alto dell'intera rete; inoltre, sappiamo già che il carrello rimane in attesa durante il tempo di ciclo, infatti se aumentassimo solamente il numero di carrelli il throughput rimarrebbe invariato mentre il costo per l'acquisto, l'istallazione e le spese di gestione aumenta. Per eliminare il bottleneck, si possono fare 2 cose: migliorare il task producePart in modo che diminuisca il tempo necessario a produrre una componente; aggiungere una macchina in modo che nello stesso intervallo di tempo precedente vengano prodotte 2 componenti invece di una; entrambe le soluzioni hanno come conseguenza l'aumento del throughput. Simuliamo i 2 casi:

1. Ottimizzando la producePart e facendo calare il suo delay da 10 UT a 9 UT (un piccolo miglioramento, poco costoso), a fine simulazione otteniamo 98 componenti, 396 task eseguiti e un thoughput di 0.098. StorePart è di nuovo nella eventList.
2. Aggiungendo una nuova macchina (inserendo un token in più nel posto m1), a fine simulazione abbiamo 178 componenti prodotte (sempre in 1000 UT), 712 task eseguiti e un throughput di 0.178. Nella eventList ci sono ben 2 eventi storePart.

Chiaramente la seconda scelta è quella che aumenta maggiormente il throughput (per la precisione, raddoppia), ma, per stabilire qual'è la decisione migliore, vanno tenuti in considerazione i costi legati alla nuova macchina e la domanda relativa al prodotto (bisogna evitare di raggiungere la saturazione). Ma qual è la configurazione ottimale? Se aggiungendo altre macchine, il bottleneck persiste, probabilmente il carrello in alcuni momenti sarà ancora in attesa di ricevere una componente e quindi il bottleneck si trova sulla macchina; se invece scompare o si sposta nel carrello significa che abbiamo già raggiunto il throughput massimo per quel numero di carrelli. Quindi analizzando le singole macchine e i singoli carrelli (il metodo *print* fornisce una stampa differenziata per ogni token) dobbiamo trovare la configurazione per cui, dato un numero di carrelli,

il numero di componenti prodotte da ogni macchina è massimo che, di conseguenza, elimina il bottleneck.

Provando diverse combinazioni, la situazione ottimale con 1 solo carrello è data con 5 macchine: a fine simulazione, ogni macchina ha prodotto 89 componenti (in totale 445), ossia il numero non è diminuito rispetto alla configurazione con una macchina, il che implica che le 5 macchine lavorano come se fossero indipendenti tra di loro, mentre in realtà sono accomunate dallo stesso carrello. Se aumentassimo ulteriormente il numero di macchine, diminuirebbe il numero di componenti prodotte da ciascuna macchina, perché il carrello non riuscirebbe a svuotarsi in tempo e farebbe attendere le macchine sulla storePart (che inizierebbe a comparire tra gli eventi pendenti). Quindi la configurazione più efficiente è quella con un rapporto 5:1 tra macchine e carrelli: infatti con 2 carrelli la configurazione ottimale richiede 10 macchine; con 3 carrelli, 15 macchine; e così via.

Attraverso la simulazione possiamo ricavare molte informazioni sul sistema, sta all'utente/addetto creare le classi dei token con le informazioni utili allo studio del sistema; in questo esempio ci siamo focalizzati sul numero di componenti prodotte, ma sono possibili altre soluzioni.

8.2 Supermarket

Il secondo esempio avanzato che abbiamo affrontato durante lo sviluppo dell'ambiente di simulazione, è un processo che riguarda una versione semplificata di un supermercato:

- Il supermercato contiene 5 tipi diversi di prodotti (identificati da un codice da 1 a 5) e inizialmente sono disponibili 10 unità per tipo.
- Un nuovo cliente entra ogni 10 UT e desidera comprare da 1 a 3 unità di un tipo di prodotto scelto casualmente; se non vi sono abbastanza unità prende quelle che trova; se il tipo richiesto è esaurito, il cliente esce dal supermercato, se invece ha trovato almeno 1 unità, si dirige alla cassa per pagare.
- Il supermercato ha due casse con coda in comune.
- Quando il tipo di un prodotto viene esaurito, esso viene rifornito in 20 UT.

Questo è un esempio di business process senza interazioni B2B (come l'esempio precedente) e con una specifica vaga, soprattutto per quanto riguarda le tempistiche: non si sa quanto impiega il cliente a selezionare il prodotto e a pagare alla cassa ("dopo un certo tempo"), quindi dobbiamo inserire dei valori temporali ragionevoli per fare in modo che la simulazione sia realistica.

Possiamo, inoltre, fare assunzioni per semplificare il lavoro dell'utente che dovrà scrivere i moduli per implementare le classi specifiche del modello: possiamo assumere che un cliente impiega 5 UT per selezionare il tipo di prodotto e impiega altri 5 UT per accodarsi e pagare alla cassa. In un caso reale, ogni azione svolta dall'utente ha le sue tempistiche: ingresso, scelta del prodotto, accodamento e pagamento (che dipende dalla quantità di oggetti acquistati); spesso nei casi studio vengono inseriti dei valori che derivano da analisi probabilistiche e valori medi; inoltre possono verificarsi situazioni che modificano il normale flusso del processo, ad esempio il cliente potrebbe ripensarci e posare un prodotto e tornare indietro. Quindi le assunzioni fatte servono semplicemente a semplificare il lavoro dell'utente finale, anche se l'ambiente di simulazione è in grado di simulare il supermercato nei minimi dettagli; questo perchè il nostro scopo non era quello di sviluppare un esempio dettagliato, ma di sviluppare un ambiente di simulazione che funzionasse per qualunque processo di business B2B.

8.2.1 Process model

Per costruite il modello, dobbiamo individuare i posti, le transizioni (con eventuali guardie e azioni), i links e i tokens della rete. A differenza degli scorsi esempi, qua non abbiamo un circuito che si chiude: un cliente arriva, prende qualcosa, paga ed esce; questa sequenza si ripete per ogni cliente che arriva nel supermercato, ogni 10 UT.

Il primo passo consiste nell'individuare le entità presenti: il prodotto ha un codice associato ed è disponibile fino ad una certa quantità, oltre la quale, viene innescata l'attività di rifornimento; pertanto creeremo una classe di token chiamata Type che rappresenta un insieme di prodotti (dello stesso tipo) con il relativo codice e quantità disponibile. La seconda entità che possiamo individuare è il cliente: egli svolge una serie di attività e sceglie il tipo di prodotto e la quantità da acquistare; pertanto creeremo una classe Client che rappresenta il cliente.

Individuate le entità coinvolte, passiamo ai posti e alle transizioni: ogni 10 UT arriva un cliente; questo implica che ci sia un task di tipo IntroduceClient che immette il cliente nella rete e questo tipo di task viene detto generativo, perché ogni 10 UT genera un token di tipo Client che viene immesso nella rete, però, a sua volta anche introduceClient necessità di un token per scoccare che non può essere di tipo Client, visto che il cliente verrà generato in seguito e conterrà delle informazioni specifiche. Quindi siamo in presenza di un task che deve scoccare sempre e dura 10 UT; pertanto nel posto che precede il task deve essere sempre presente un token slegato dal resto del processo che deve fare scoccare introduceClient e che viene reinserito nel posto "gen" da tale

transizione: creeremo quindi una nuova classe di token chiamata Empty perché di fatto non ha alcuna informazione al suo interno e non deve essere propagato all'interno della rete, serve solo a fare scoccare tale task. Pertanto verrà creato un posto "gen" in cui è presente almeno 1 token (altrimenti i clienti non arriverebbero mai) che precede tale transizione, seguita da un posto "c1" in cui verrà messo il token di tipo Client generato.

A questo punto il cliente va alla ricerca di un tipo di prodotto: se la quantità disponibile è 0, esce dal supermercato, quindi creiamo una transizione di classe ClientExits; altrimenti, prende la quantità desiderata (da 1 a 3 unità) oppure la quantità presente, se la quantità disponibile è inferiore rispetto a quella desiderata, ma non nulla; creiamo quindi una transizione di classe ClientTakesProducts. Questa situazione si traduce in una scelta: bisogna introdurre una guardia che verifica la quantità disponibile e lascia passare il token solo se la se tale quantità è diversa da 0; questa guardia verrà messa su clientTakesProducts in modo da attivare clientExits quando la condizione non è soddisfatta. Se il cliente trova almeno 1 unità del prodotto desiderato, viene immesso un token di tipo Client in un posto "c2" che segue la transizione clientTakesProducts.

In seguito il cliente, se non è già uscito, si dirigerà alla cassa quindi creeremo una transizione di classe ClientGoesToCheckout che dura 5 UT: questo valore, per semplicità, racchiude anche il tempo necessario anche a trovare e prendere il prodotto desiderato; pertanto le transizioni clientTakesProducts e clientExits avranno delay pari a 0 UT. La transizione clientGoesToCheckout emetterà un token di tipo Client in un posto "c3".

A questo punto il cliente si accoda e paga, quindi abbiamo una transizione di tipo ClientPays di durata 5 UT, ma da sola non basta: la specifica parla di due casse, quindi potrebbe verificarsi il caso in cui un entrambe le casse sono occupate e il cliente attende in coda; questo implica l'esistenza di un posto "checkout" (cioè cassa) che contiene un numero di token pari al numero di casse attive nel supermercato e rende le casse disponibili sempre ad eventuali clienti. Pertanto inseriremo due tokens di tipo Empty (nel nostro esempio non serve un token di tipo Checkout, ma è eventualmente possibile usare un token specifico) in tale posto che sarà collegato alla transizione clientPays per fare in modo che le 2 casse servano i clienti; inoltre, per fare in modo che le casse siano sempre disponibili, questi token devono restare nello stesso circuito, esattamente come nel caso del circuito composto dal posto "gen" e dalla transizione introduceClient che si autoalimenta; quindi aggiungeremo un collegamento da clientPays a "checkout".

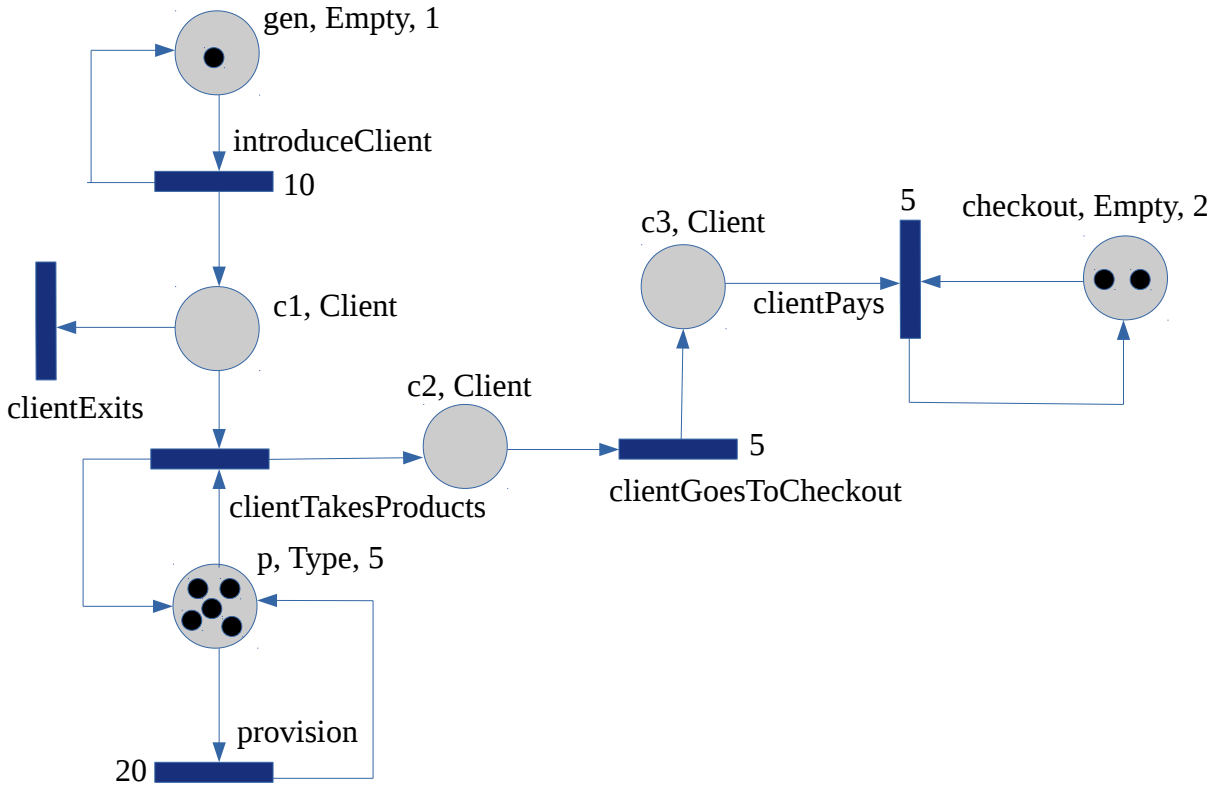
Infine va gestita la disponibilità dei prodotti: la transizione clientTakesProducts deve poter leggere la quantità disponibile di quel tipo di prodotto, quindi deve avere accesso ad una specie di inventario del supermercato in cui ci sono le informazioni relative ai prodotti. Possiamo rappresentare l'inventario con un posto "p" che contiene 1 token per prodotto e che è collegato a

clientTakesProduct in modo che la guardia della transizione possa esaminare tutti i token di tipo Type e prelevare quello desiderato dal cliente. Poichè il token di quel tipo di prodotto non deve sparire, ma deve essere sempre disponibile per la consultazione, viene reimmesso nello stesso posto "p" da tale transizione. Inoltre, quando uno dei prodotti presenti in "p" ha quantità disponibile pari a 0, deve essere innescata una transizione di classe Provision che rifornisce il supermercato di 10 nuove unità di quel tipo di prodotto in 20 UT; tale transizione deve controllare di continuo le quantità disponibili, quindi verrà inserita una guardia che blocca la transizione se la quantità è maggiore di zero; anche in questo caso, avremo un circuito autoalimentato composto da "p" e da provision.

L'ultimo passo consiste nell'inserire le azioni nella rete: quando viene eseguita clientTakesProducts, bisogna aggiornare la quantità disponibile sottraendo la quantità effettivamente presa dal cliente; pertanto aggiungiamo un'azione in quella transizione. Un'altra azione necessaria va inserita su Provision: infatti, a seguito di un rifornimento, va aggiornata la quantità disponibile per il tipo di prodotto appena rifornito. Nel nostro modello non inseriremo altre azioni, ma per completezza, bisognerebbe inserire 2 azioni, rispettivamente su clientExits e clientPays che tolgono i 2 token di tipo client dalla rete perché il cliente, in entrambi i casi, esce dal supermercato; altrimenti nella stampa finale verrebbero visualizzate anche le entità non più presenti della rete il che potrebbe anche rivelarsi utile per analizzare certe situazioni. Un'altra semplificazione riguarda clientPays: il pagamento dovrebbe essere proporzionale al numero di articoli acquistati: il simulatore legge l'oggetto T, che viene definito con un delay fisso, quindi modificare la durata della transizione durante la simulazione non è possibile; per risolvere il problema, è possibile inserire un posto "n" in uscita da clientPays con due transizioni di output; la prima è la stessa clientPays, in cui ci sarà una guardia che blocca la transizione se la quantità presa dal cliente è pari a 0 (la prima esecuzione è garantita perché altrimenti il cliente sarebbe già uscito) ed ha un'azione che decrementa di una unità tale quantità, mentre l'altra transizione è identica a clientExits (potremmo addirittura usare la stessa transizione) che viene eseguita quando la guardia blocca clientPays. Nel nostro esempio abbiamo preferito non aumentare la complessità della rete, ma è doveroso sottolineare che ciò era possibile.

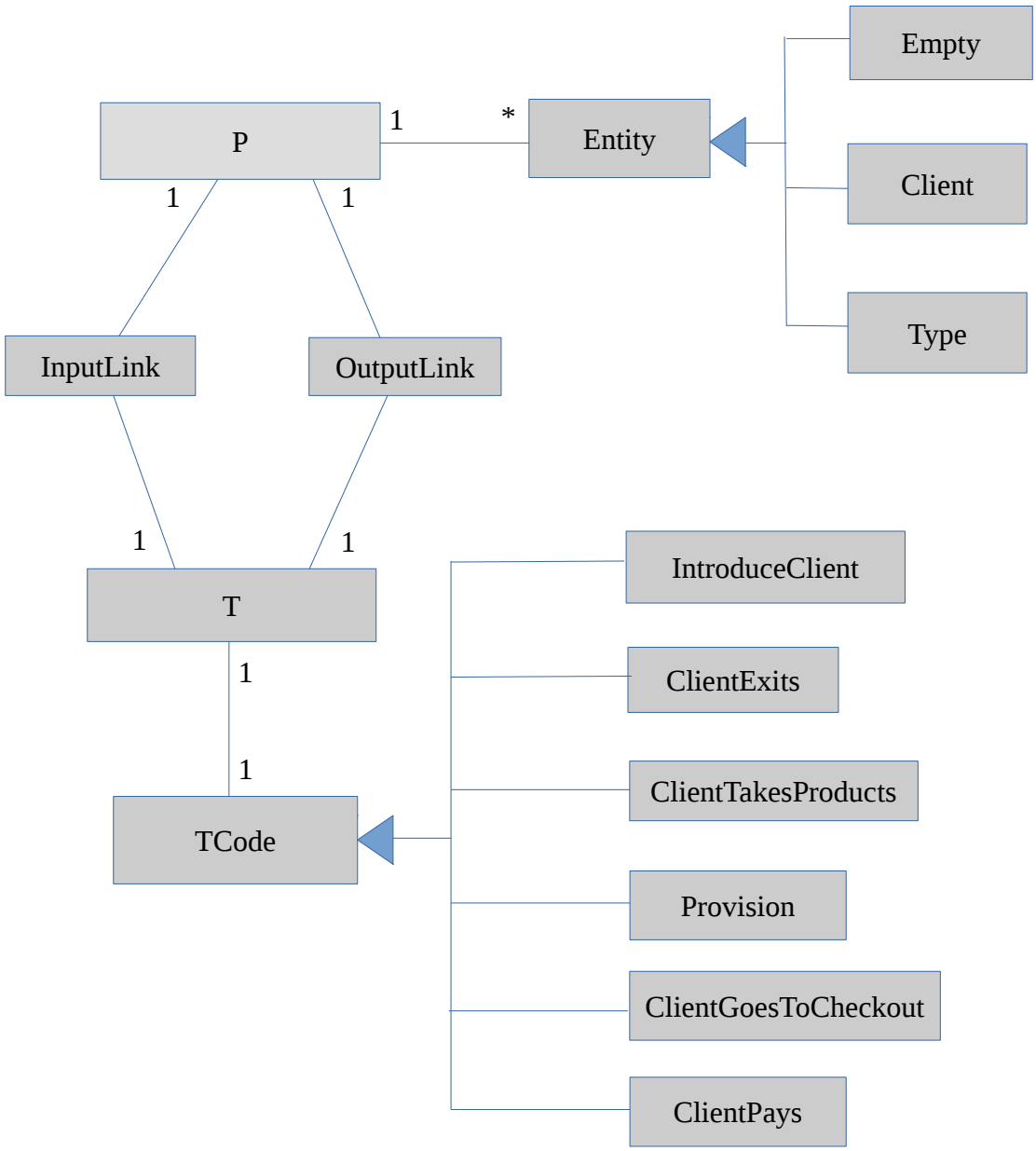
Ora che la rete è definita, non ci resta che inserire i tokens: avremo sicuramente 1 token in "gen" che consente a introduceClient di scoccare; 5 tokens in "p" che rappresentano i 5 tipi di prodotto presenti nel supermercato; infine avremo 2 tokens in "checkout" che rappresentano le 2 casse e quindi possono far scoccare 2 clientPays contemporaneamente.

Quindi il process model risultante è il seguente:



8.2.2 Il modello informativo

Avendo individuato le entità e i task del sistema, il modello informativo è facilmente ricavabile, aggiungendo le classi specifiche necessarie per simulare il sistema:



8.2.3 Applicazione dell'ambiente

La classe Main, che viene utilizzata per costruire la rete, inserire i tokens, far partire la simulazione e stampare i risultati (come negli esempi precedenti), è la classe principale; riportiamo qui di seguito la costruzione della rete:

```
Net net = new Net("Supermarket1", "supermarket");

net.p("gen", "Empty").p("c1", "Client").p("p", "Type")
      .p("c2", "Client").p("c3","Client").p("checkout", "Empty")
      .t("IntroduceClient", 10).t("ClientExits", 0)
      .t("ClientTakesProducts", 0).t("Provision", 20)
      .t("ClientGoesToCheckout", 5).t("ClientPays", 5)
      .l("gen->IntroduceClient").l("IntroduceClient->gen")
      .l("IntroduceClient->c1").l("c1->ClientExits")
      .l("c1->ClientTakesProducts").l("ClientTakesProducts->c2")
      .l("ClientTakesProducts->p").l("p->ClientTakesProducts")
      .l("p->Provision").l("Provision->p")
      .l("c2->ClientGoesToCheckout").l("ClientGoesToCheckout->c3")
      .l("c3->ClientPays").l("ClientPays->checkout")
      .l("checkout->ClientPays");
```

Segue l'inserimento dei tokens, lo start della simulazione e la stampa dei risultati:

```
nets.add(net);

Simulator sim = new Simulator(nets);

try {
      sim.putToken("gen", new Empty(), net);

      for(i=0; i<checkouts; i++) {
            sim.putToken("checkout",new Empty(), net);
      }

      for(i=0; i<products; i++) {
            sim.putToken("p",new Type(), net);
      }

} catch (Exception e) {
      e.printStackTrace();
}

sim.run(500);

sim.print();
```

Dove checkouts e products rappresentano il numero di casse attive e di tipi prodotti acquistabili nel supermercato.

Il passo successivo consiste nello scrivere le classi dei tokens: Empty, Client e Type. La prima classe rappresenta un token vuoto, senza informazioni, che serve solamente per fare scoccare dei task; quindi la estenderà il comportamento di Entity ma non conterrà nulla, l'unico metodo che è sempre consigliato ridefinire è il *toString* per avere una stampa dell'oggetto adatta alle proprie esigenze.

La classe Client invece dovrà contenere tutte le informazioni riguardanti lo specifico cliente: avremo un attributo statico che rappresenta la quantità massima di un determinato prodotto che un cliente può richiedere (nel nostro caso è pari a 3) e avremo un attributo che rappresenta la quantità richiesta e un altro indica il codice del tipo di prodotto desiderato; quest'ultimi valori vengono inizializzati nel costruttore della classe creando un oggetto Random e chiamando il suo metodo *nextInt* a cui vengono passate le 2 variabili statiche che maxQuantity e typeCode (indica il valore dell'ultimo codice assegnato agli oggetti Type) e restituisce un intero random minore o uguale al valore passato come parametro; Ci sarà inoltre un altro attributo intero che indica la quantità effettivamente presa dal cliente. Per quanto riguarda i metodi, ci serviranno i getters degli attributi che indicano il tipo di prodotto e la quantità desiderata, mentre la quantità presa necessita solamente del metodo setter per settare tale attributo dato che, poiché non sono ammessi ripensamenti da parte del cliente, l'aggiornamento dell'inventario può essere fatto dentro a clientTakesProducts e non durante clientPays (è una semplificazione, ma sarebbe comunque possibile perché il token Client ha il codice prodotto al suo interno). Anche qui, si consiglia di ridefinire il metodo *toString* per stampare i valori dei vari attributi; notare che l'attributo statico non ha metodi getter o setter perché è lo stesso per tutte le istanze della classe e viene deciso dall'utente durante la definizione della classe.

La classe Type rappresenta i tipi di prodotti del supermercato: avrà quindi un attributo intero, statico e progressivo che serve per la generazione dei codici (typeCode); un attributo intero che rappresenta il codice di quel tipo di prodotto e un intero che indica la quantità disponibile; eventualmente anche un intero che indica la quantità totale acquistata dai clienti per comparare tra di loro i tipi di prodotti ed effettuare analisi atte a migliorare il rifornimento del supermercato. Anche in questo caso avremo metodi getters e setters e verrà ridefinito il *toString*; inoltre servirà un metodo che decrementa la quantità disponibile a seguito di un acquisto (es eventualmente incrementa la quantità totale venduta), mentre per il rifornimento si può semplicemente usare il setter sull'attributo che rappresenta la quantità disponibile.

Infine l'utente deve scrivere le classi dei task del processo: IntroduceClient, ClientExits, ClientTakesProducts, Provision, ClientGoesToCheckout e ClientPays. La prima classe, rappresenta un task generativo: non ha la guardia, ma ha un'azione che genera un nuovo token di tipo Client, lo

assegna alla rete (è la stessa del token di tipo Empty prelevato dal posto di input) e lo aggiunge alla lista dei token che finiranno nei posti di output (assieme al token di tipo Empty che tornerà nel posto “gen”).

ClientExits, nel nostro caso, sarà una classe che estende TCode (come tutte le altre che rappresentano un task), ma non ha né guardie né azioni, quindi possiamo definirla attraverso una classe vuota (non vogliamo eliminare i token dalla rete una volta usciti) che verrà eseguita tutte le volte in cui la guardia è attiva su clientTakesProducts. Essa, invece, avrà sia una guardia che un’azione: la guardia consulterà sia i tokens Client proveniente dal posto “c1”, sia i tokens Type provenienti dal posto “p” e confronterà il codice del prodotto richiesto dal cliente con quello dei token di tipo Type e se c’è almeno un’unità disponibile di quel tipo di prodotto, abiliterà la transizione, altrimenti la bloccherà (e di conseguenze verrà abilitata clientExits). L’azione invece aggiornerà la quantità disponibili sottraendole la quantità effettivamente presa dal cliente che può essere pari alla quantità richiesta o pari alla quantità disponibile (se questa è minore di quella richiesta).

La transizione provision ha sia la guardia che l’azione: la guardia controllerà tutti i tipi di prodotto attraverso i loro token e abiliterà la transizione per i token che hanno quantità disponibile pari a 0, in modo il supermercato venga rifornito di quel tipo di prodotto. L’azione ,invece, aggiorna la quantità disponibile settandola con un certo valore (ad esempio di 10 unità) che permette ai clienti successivi di acquistare quel prodotto.

Le transizioni ClientGoesToCheckout e ClientPays sono entrambe vuote perché abbiamo spostato tutta la logica e i dati nelle transizioni precedenti e non sono richiesti dati specifici agli eventi che accadono durante queste transizioni (ad esempio non viene richiesto di calcolare il tempo medio di accodamento e così via).

Notare che, in questo esempio, ci sono più entità con diverse transizioni in comune: è compito delle guardie selezionare tra tutti i tokens dei posti di input, solamente quelli che servono per abilitare la transizione; questi poi saranno propagati dall’azione che eventualmente li modificherà; infine verranno immessi nei posti di output.

8.2.4 Risultati ottenuti

In questo esempio, abbiamo simulato il sistema con 5 diversi tipi di prodotti, 2 casse attive e un nuovo cliente ogni 10 UT, semplificando notevolmente un possibile sistema reale che avrebbe

numeri sicuramente più grandi; ciò nonostante, l'importante era impostare il sistema per poterlo simulare in seguito con numeri più fedeli al caso reale. Il token su "gen invece deve rimanere 1 solo, perché, questa è una limitazione della rete impostata dall'utente, perché nel caso ci fossero 2 tokens, arriverebbero insieme 2 clienti che potenzialmente potrebbero chiedere lo stesso prodotto, quindi la guardia controllerebbe lo stesso prodotto e abiliterebbe entrambe le transizioni, ma le rispettive azioni decrementerebbero la quantità due volte, rischiando di giungere a valori minori di zero che non sono accettabili. Questo limite deriva dal fatto che stiamo simulando a tempo discreto, ma anche nella situazione reale i clienti prenderebbero i prodotti "accodati" e se sono in esaurimento, fanno a gara a chi arriva prima. Per affollare il supermercato, si consiglia quindi di diminuire il delay di introduceClient, in modo da far entrare i clienti più di frequente.

Simulando il sistema con i numeri visti finora (un piccolo supermercato) e inserendo un endTime pari a 500, otteniamo, a fine simulazione, i seguenti risultati: sono stati completati 198 task, sono stati acquistati 88 prodotti e solamente 3 clienti sono usciti insoddisfatti perché il prodotto che cercavano era esaurito (il rifornimento è arrivato dopo la loro dipartita); inoltre, nella eventList, abbiamo 2 eventi, uno contiene la transizione introduceClient e l'altro la transizione ClientGoesToCheckout che sono 2 tra le transizioni a durata maggiore (è importante scoprire se ci sono bottleneck nella rete). La simulazione ci dice anche quante unità sono state vendute per ciascun tipo di prodotto, in questo caso, il prodotto più venduto è stato quello con il codice 2, con 23 unità vendute; anche questo è un dato importante, sia per aiutare a gestire in modo efficiente il sistema dei rifornimenti (ad esempio dando priorità ai prodotti più venduti), sia per incentivare l'acquisto dei prodotti meno acquistati (ad esempio applicandoci uno sconto). Un altro dato importante è il numero dei clienti entrati e serviti alle casse: sono entrati 50 clienti; ne sono usciti 3, ne sono stati serviti 46; uno si stava dirigendo alle casse quando si è raggiunta la fine della simulazione. Cosa succede se chiudiamo una delle due casse? I numeri rimangono all'incirca gli stessi (il generatore random impedisce la ripetibilità esatta), quindi in realtà in tale situazione non conviene tenere 2 casse aperte, ma una sola; questa è una piccola ottimizzazione, che, se non cambiano le tempistiche, permette di risparmiare il 50% sui costi legati alle casse (commessi, corrente elettrica per far girare il rullo, ecc..).

Proviamo ora a simulare un supermercato di medie dimensioni, con 30 diversi tipi di prodotto, rifornimento iniziale a 50 e quantità massima acquistabile da un cliente 15; manteniamo solamente 2 casse aperte. Notiamo che, all'aumentare del numero di prodotti, è più improbabile che clienti consecutivi scelgano lo stesso prodotto, infatti clientExits capita molto di rado o non capita affatto perché, una volta esaurito, si riesce a rifornire il supermercato prima dell'arrivo del prossimo cliente richiedente quel prodotto. Il numero di clienti serviti questa volta è 97, circa il doppio rispetto alla

situazione precedente, la cosa è ragionevole perché la frequenza di clienti è raddoppiata. Non abbiamo inserito metriche sui tempi di attesa delle casse, ma anche in questo caso non servono più di 2 casse aperte, anzi ne basterebbe 1 sola, poiché le transizioni con delay diverso da 0, durano tutte 5 UT si crea una catena alla cassa in cui il commesso serve i clienti costantemente, tranne nei rari casi in cui un cliente esce dal supermercato, in cui il commesso riposa 5 UT. Possiamo concludere, anche in questo caso, che possiamo ridurre il numero di casse aperte e il rapporto clienti soddisfatti/insoddisfatti è accettabile.

Infine proviamo a testare un supermercato di grandi dimensioni con un nuovo cliente ad ogni UT, 200 diversi tipi di prodotto, quantità disponibile pari a 150 (uguale al rifornimento) e quantità massima prelevabile pari a 50 unità; 2 casse aperte. I risultati di questa simulazione sono interessanti: non ci sono problemi nel rifornimento data la grande varietà di prodotti (provision non viene quasi mai lanciata), ma ci sono evidenti problemi nella clientPays, in cui le code alle casse rallentano moltissimo il flusso di clienti: infatti solo 200 clienti su 500 riescono ad essere serviti; la cassa è un collo di bottiglia. La controprova la si ha diminuendo il numero di casse aperte: con 1 cassa aperta i clienti serviti scendono a 100. Per eliminare il collo di bottiglia, servono almeno 5 casse operative: in questo modo vengono serviti in modo continuo tutti i clienti che entrano nel supermercato.

Sono ovviamente possibili infinite combinazioni, e si può complicare il modello del supermercato per avere una simulazione sempre più fedele al caso reale; ma da questi pochi esempi possiamo ricavare alcune nozioni: il numero di casse da aprire è proporzionale alla frequenza di ingresso dei clienti; quindi a seconda dell'orario e del giorno, in base ai dati statistici del supermercato specifico, è possibile stabilire un programma di turni e apertura delle casse, per ottimizzare il servizio rivolto ai clienti e ridurre il costo della gestione. Un'altra nozione riguarda la gestione dell'inventario: supponendo la scelta del cliente completamente casuale, all'aumentare della varietà di prodotti diminuisce la probabilità che venga richiesto un prodotto già esaurito; questo implica che si può abbassare la quantità di prodotti disponibili se la varietà è alta; infatti in una situazione reale, se manca quel tipo di prodotto, ma la varietà è alta, è probabile che il cliente si accontenterà di scegliere un altro prodotto simile e non uscirà dal supermercato a mani vuote.

Sono possibili altri tipi di analisi, a patto di definire le classi con i dettagli giusti; ad esempio si potrebbe studiare, lato cassa, il tempo di attesa medio in coda dei clienti; associare un peso ad ogni prodotto per influenzare il generatore randomico, pesando di più prodotti tipicamente più gettonati rispetto ad altri; studiare il rifornimento perfetto trovando la soglia giusta sotto la quale viene fatto partire il rifornimento e così via. Il fine ultimo è sempre quello di ottimizzare l'utilizzo delle risorse che non devono rimanere in stati d'attesa, se non per brevi periodi; ma anche minimizzare i costi di

gestione, i tempi di rifornimento e i turni alle casse; producendo un risparmio non indifferente che influisce sul bilancio aziendale.

Capitolo 9

Esempio con processo B2B

9.1 Buyer-Seller

L'ultimo esempio di applicazione dell'ambiente di simulazione è un processo di business B2B completo: il sistema è composto da 2 buyer (un acquirente) e 3 seller (i venditori); buyer e seller fanno parte di organizzazioni diverse, ognuna con il proprio sistema informativo. Un buyer può inviare una richiesta di un prodotto ai seller con il quale ha un rapporto di partnership (supponiamo con tutti e 3), i quali riceveranno la richiesta del buyer, genereranno un'offerta e la invieranno al buyer. Egli attenderà l'arrivo delle offerte di tutti i partner e poi deciderà di accettare un'offerta e rifiutare tutte le altre in base ad un qualche criterio (ad esempio, accetta quella a prezzo più basso); quindi invia un messaggio a quegli stessi partner con la risposta positiva o negativa.

Quindi l'ambiente simulerà le interazioni tra le diverse organizzazioni, trasportando i token da una rete all'altra (ogni organizzazione ha la propria rete). Per far questo, non bastano il process model e i modello informativo, ma serve anche un modello collaborativo. In questo caso avremo dei modelli abbastanza semplici, ma l'ambiente è in grado di simulare anche sistemi più complessi.

9.1.1 Il modello collaborativo

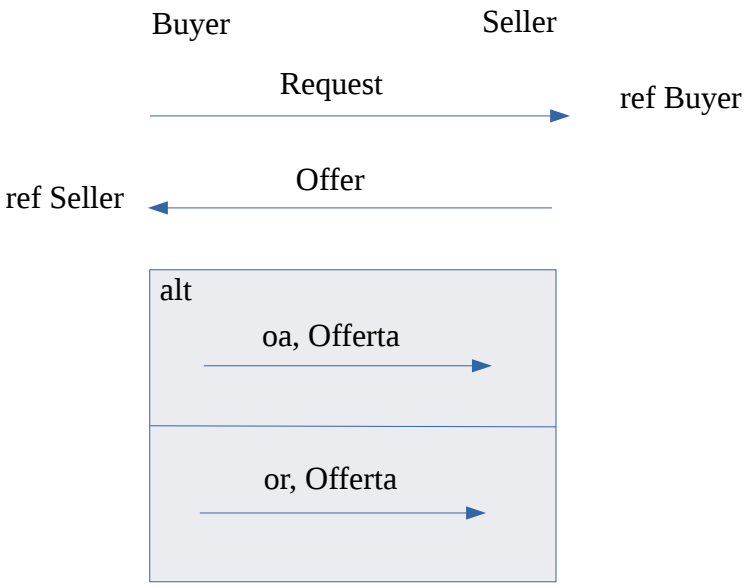
La presenza di più organizzazioni nel sistema implica la necessità di definire dei protocolli di comunicazione, in cui si decide esse dovranno interagire tra di loro. Dai dati a disposizione, si deduce che l'initiator della comunicazione è sempre il buyer, mentre il seller riveste il ruolo di follower. Il buyer invia un messaggio al seller contenente un oggetto Request che ha al suo interno, tutte le informazioni necessarie al seller per poter generare un'offerta e associarla a quella determinata richiesta di quello specifico buyer.; questo tipo di interazione si dice generativa, perché ha come conseguenza la generazione di una nuova entità nel sistema informativo dell'altra organizzazione.

A questo punto il seller invia l'offerta generata a tale buyer con un messaggio contenente un oggetto Offer con tutte le informazioni necessarie al buyer per prendere la decisione di accettare o rifiutare

l’offerta. Poichè il buyer deve attendere le offerte di tutti i seller associati, genererà un’entità offerta nel proprio sistema informativo per immagazzinare tale informazione; quindi anche questa è un’interazione generativa.

Infine, una volta giunte tutte le offerte dai seller, il buyer prende la sua decisione: accetta 1 sola offerta e rifiuta tutte le altre e invierà un messaggio a quei seller con l’esito della sua decisione che verrà memorizzato da quest’ultimi. Questa interazione non genera una nuova entità nel sistema informativo di un’altra organizzazione, ma va a modificare un entità già presente al suo interno; in questo caso si parla di interazione modificativa; pertanto a tale oggetto verrà aggiunto “oa” che indica un’offerta accettata, oppure “or” che indica che l’offerta è stata rifiutata.

Di conseguenza, il modello collaborativo tra i 2 tipi di organizzazioni presenti è il seguente:



9.1.2 Process model

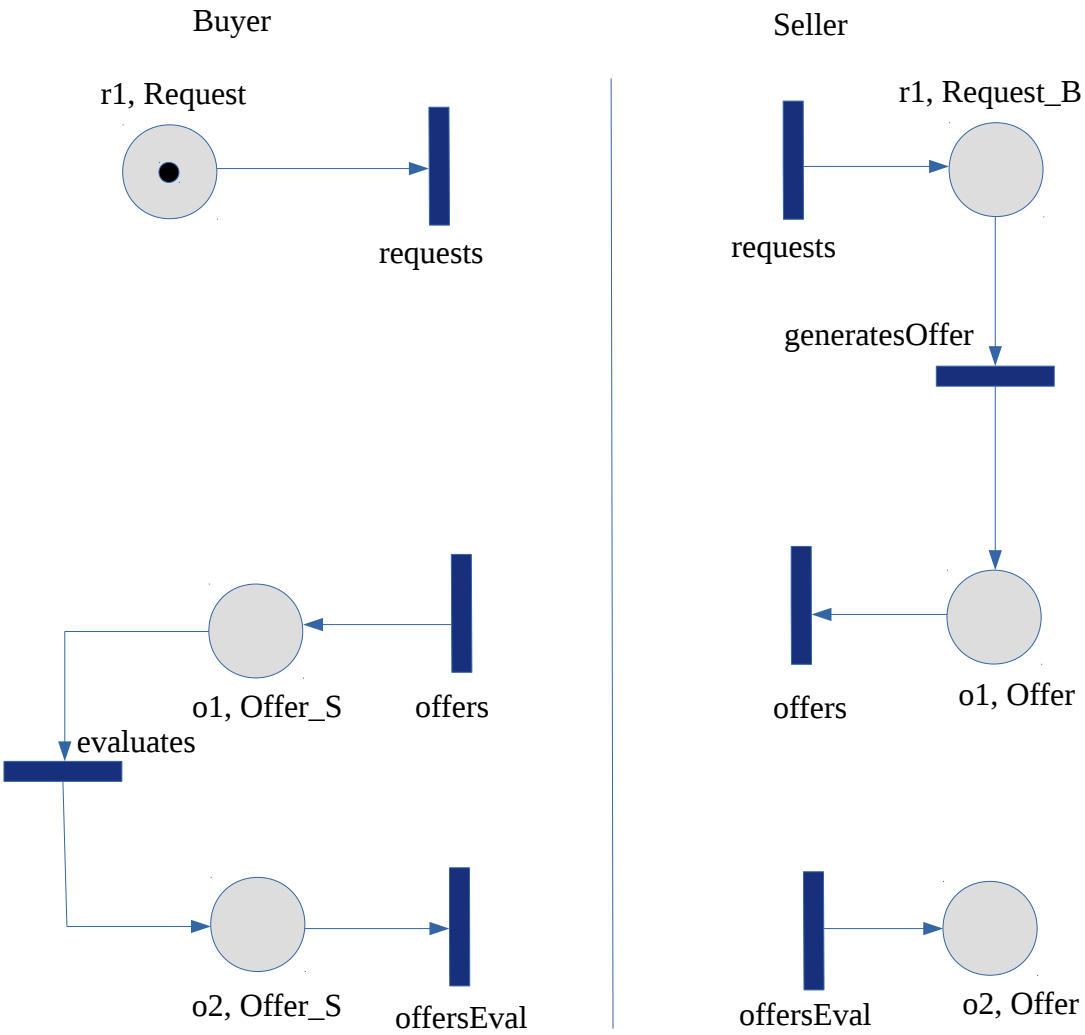
Come negli esempi precedenti, cerchiamo di individuare gli elementi costitutivi della rete, ricavandoli dalla specifica e aiutandoci con il modello collaborativo. Partiamo dal buyer perché è l’initiator della collaborazione: creiamo quindi un posto “r1” in cui verrà inserito il token che farà partire l’intero processo; il token sarà di tipo Request e questo posto sarà collegato ad una transizione di tipo Requests che permette al buyer di inviare una richiesta ai suoi partner. Questa transizione non ha alcun link di output all’interno della rete del buyer, perché non è una transizione

comune, ma è una transizione di tipo send che invierà la richiesta del buyer (quindi il token di tipo Request) sotto forma di messaggio a tutti i partners atti a ricevere tale richiesta; pertanto requests è una transizione comune tra le due reti che funge da collegamento e fa passare il token da una rete all'altra. Ogni ricevente riceverà un messaggio con il token nella propria transizione requests che, però, è di tipo receive e, inoltre, siamo in presenza di un'interazione generativa; quindi la transizione ricevente requests deve generare una nuova entità nel sistema informativo del Seller per immagazzinare tutte le informazioni che riguardano la Request; verrà emesso un token di tipo Request_B che oltre alle informazioni contenute nel messaggio, conterrà anche informazioni utili per associare la richiesta al mittente; la requests non ha né guardia né azione.

La requests del Seller emetterà un token di tipo Request_B in un posto "r1" nella rete di quest'ultimo che a questo punto deve generare un'offerta: seguirà quindi una transizione di tipo generatesOffer che è senza guardia, ma invece ha un'azione che genera un nuovo tipo di token che rappresenta un'offerta: Offer. Notare che il posto "r1" esiste già nella rete del buyer, ma non del seller, quindi è possibile nominarlo con lo stesso nome; di fatto in una situazione reale, buyer e seller si accordano solo sulle entità globali che verranno scambiate secondo il modello collaborativo, ma non condividono la propria rete quindi l'omonimia è altamente probabile. La transizione emetterà il token di tipo Offer in un posto "o1" e seguirà una transizione che avrà il compito di inviare l'offerta al buyer che aveva fatto la richiesta correlata, ossia una transizione di tipo Offers: essa creerà un messaggio in cui inserirà il token di tipo Offer; anche questa è una transazione di tipo send che farà passare il token dalla rete del seller a quella del buyer. Egli infatti avrà una transizione omonima che sarà di tipo receive ed essendo anche questa un'interazione generativa, offers genererà un nuovo token di tipo Offer_S che conterrà le informazioni contenute nel messaggio e verrà emesso dalla transizione in un posto "o1". A questo punto, il buyer dovrà attendere l'arrivo di un'offerta da ogni seller a cui aveva inviato la richiesta, accettare quella migliore e rifiutare le altre; quindi al posto "o1" segue una transizione di tipo Evaluates che avrà sia la guardia che l'azione: la guardia controllerà i tokens presenti in "o1" e quando saranno arrivati tutti i token (1 per ogni seller e riferenti la stessa richiesta), abiliterà la transizione; l'azione invece selezionerà l'offerta migliore e ne cambierà lo stato in "accettata", mentre a tutte le altre offerte lo stato verrà settato su "rifiutata". Evaluates emetterà i token (sempre di tipo Offer_S) delle relative offerte in un posto "o2" che precede una transizione di tipo OffersEval: questa ha il compito di inviare, ai relativi seller, la risposta all'offerta; è una transizione di tipo send che invierà un messaggio contenente il token Offer_S con l'esito della decisione del buyer. Pertanto il token di tipo Offer_S passerà nella rete del relativo seller attraverso una transizione di classe OffersEval e di tipo receive: questa implementa un'interazione modificativa, pertanto andrà a recuperare il token di

classe Offer (già presente nel sistema informativo del seller) a cui il messaggio fa riferimento e ne modifica lo stato in base alla decisione del buyer.

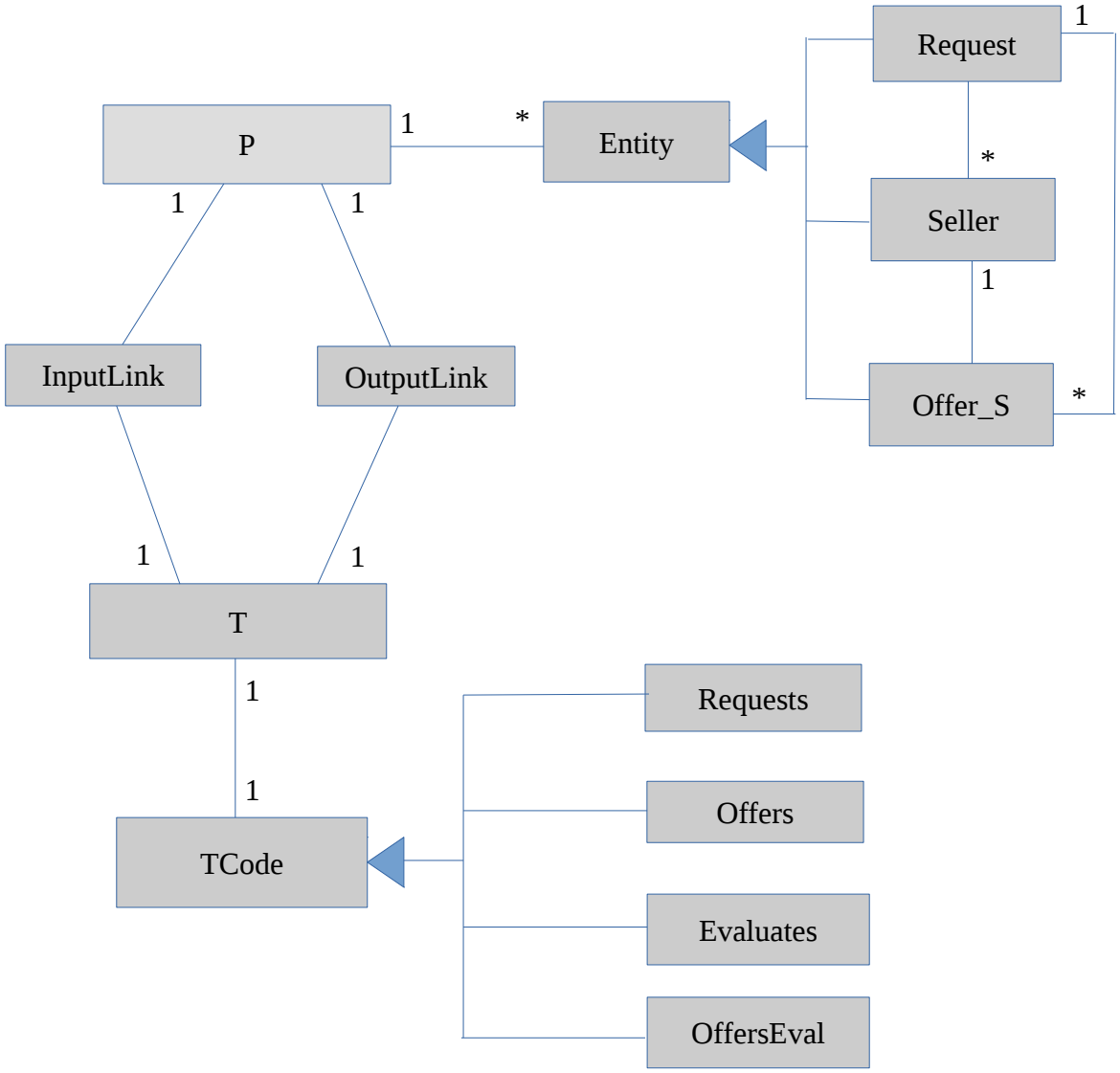
Quindi il process model risultante è il seguente:



Notiamo che non esiste un circuito chiuso, quindi dopo offersEval il processo si ferma. Infatti, non sono state definiti né i delay, né una qualche frequenza di generazione dei token su “r1” che farebbero partire il processo; questo perché, come negli scorsi esempi, ci siamo soffermati sugli aspetti funzionali della libreria in modo che potesse funzionare anche con esempi più complessi di processi di business B2B.

9.1.3 Il modello informativo

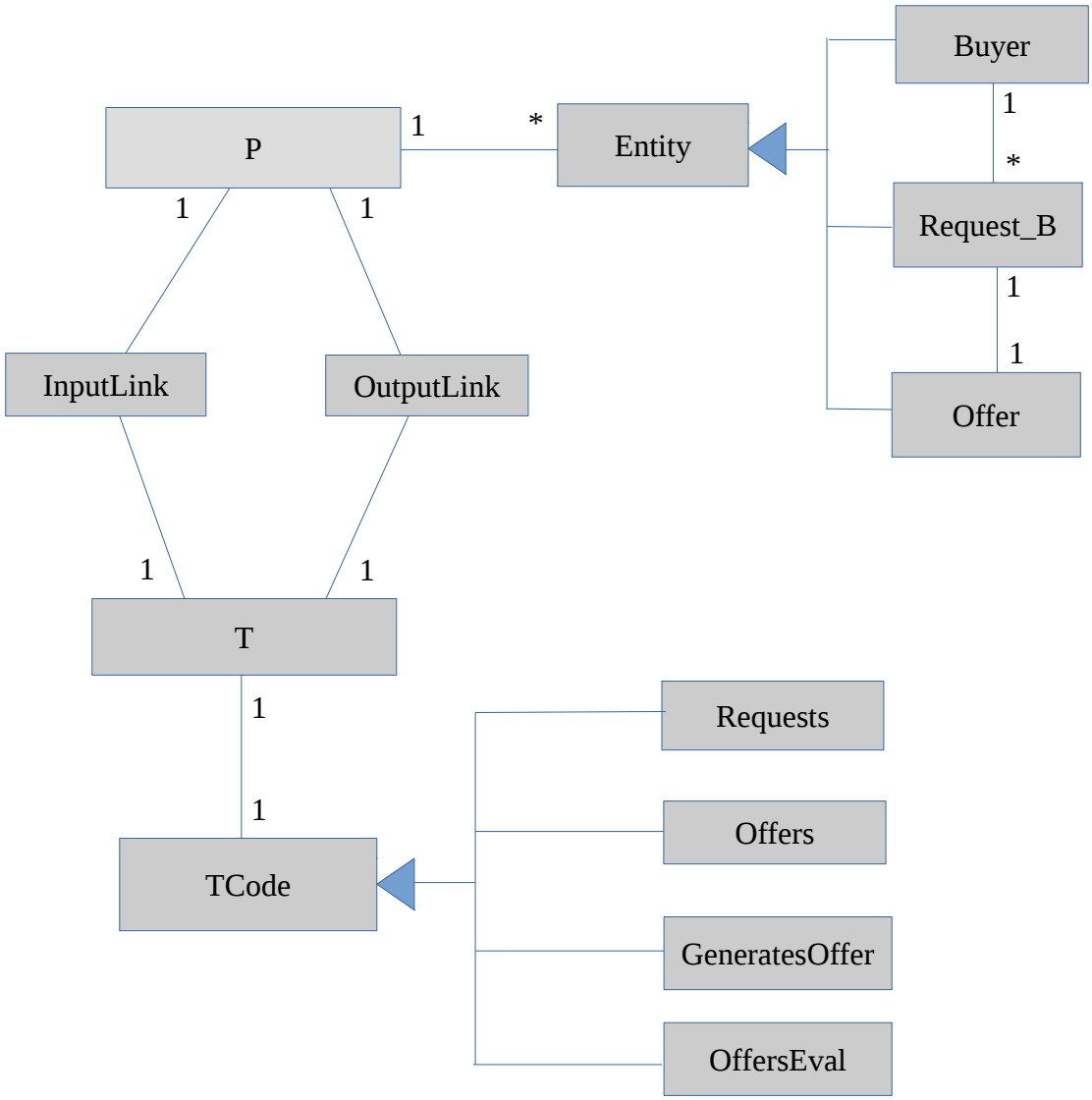
I buyer e i seller hanno sistemi informativi distinti e non condivisi; avranno in comune la parte di libreria, che riportiamo per completezza, ma il modello informativo vero e proprio riguarda le entità specifiche della rete interna dell'organizzazione che estendono la classe di libreria Entity. Partendo dal buyer, aggiungiamo le transizioni presenti nella sua rete che estenderanno TCode; e in seguito aggiungiamo i 3 tipi di token presenti nella sua rete.



In questo esempio, inoltre, è necessario specificare le relazioni tra le varie entità, in quanto sono strettamente collegate l'una con l'altra: un buyer teoricamente può fare molte richieste; una richiesta può essere inviata a tanti seller e ogni seller può ricevere più richieste dallo stesso buyer(relazione molti a molti); ad una richiesta corrispondono più offerte (pari al numero di seller a

cui è stata inviata), ma ad un'offerta corrisponde una sola richiesta (relazione 1 a molti); infine ad ogni offerta corrisponde un solo seller mentre ogni seller può (se riceve più richieste) preparare più offerte (relazione 1 a molti).

Lo stesso discorso vale per il modello informativo del seller:



Anche qui dobbiamo aggiungere i task specifici della rete del seller e specificare le relazioni tra le varie entità: Un seller può ricevere più richieste contemporaneamente; Una richiesta è associata ad un solo buyer mentre un buyer può effettuare più richieste (relazione 1 a molti); ad una richiesta corrisponde una sola offerta e viceversa (relazione 1 ad 1); il buyer e l'offerta non necessitano un collegamento diretto, per trovare l'offerta corrispondente a quel buyer bisogna navigare nel modello passando attraverso la richiesta.

9.1.4 Applicazione dell'ambiente

Dopo aver definito i 3 modelli che rappresentano il sistema, possiamo passare all'applicazione dell'ambiente, sfruttando le sue classi di libreria e definendo le classi specifiche del sistema studiato.

Per prima cosa definiamo la classe Main che costruirà le reti e inserirà i token iniziali: poiché abbiamo 2 buyer e 3 seller, potrebbe e le reti sono identiche all'interno di una categoria, conviene creare delle funzioni *buildBuyerNet* e *buildSellerNet* che costruiscono rispettivamente una rete buyer e una seller e ritornano la rete appena costruita; in modo da poter chiamare più volte queste funzioni invece di ripetere lo stesso codice più volte. La funzione *buildBuyerNet* potrebbe essere così definita:

```
Net net = new Net(name, packageName);

net.p("r1", "Request").p("o1", "Offer_S").p("o2", "Offer_S")
      .tSend("Requests", 5).tReceive("Offers", 5).t("Evaluates", 5)
      .tSend("OffersEval", 5).l("r1->Requests").l("Offers->o1")
      .l("o1->Evaluates").l("Evaluates->o2").l("o2->OffersEval");
return net;
```

Mentre la funzione *buildSellerNet* potrebbe essere definita in questo modo:

```
Net net = new Net(name, packageName);

net.p("r1", "Request_B").p("o1", "Offer").p("o2", "Offer")
      .tReceive("Requests", 5).t("generatesOffer", 5).tSend("Offers", 5)
      .tReceive("OffersEval", 5).l("Requests->r1").l("r1->generatesOffer")
      .l("generatesOffer->o1").l("o1->Offers").l("OffersEval->o2");

return net;
```

Notare che abbiamo aggiunto dei delay di 5 UT ad ogni transizione anche se non erano stati specificati nel testo; questo perché non essendoci né un circuito chiuso né un generatore di token, la simulazione terminerà quando verrà eseguita offersEval nei seller; ma, per avere maggiore chiarezza durante la lettura dei risultati, in modo che ogni fase sia ben separata temporalmente, abbiamo deciso di aggiungere tali delay. Notiamo anche le chiamate a *tSend* e *tReceive*; come detto, questo è un esempio B2B e tali transizioni rappresentano le interazioni tra le organizzazioni coinvolte.

In questo caso, sarà necessario anche collegare i vari partner tra di loro, definendo un metodo *addSeller* da qualche parte; questo metodo verrà chiamato nel main e serve per evitare di inviare le

richieste ai seller che non sono partner di un determinato buyer. Inoltre, i token iniziali sono di tipo Request e vanno inseriti nei buyer. Quindi andiamo a definire le classi delle varie entità (che estenderanno Entity), rispettando il modello informativo, per cui un buyer avrà nel suo sistema informativo dei seller e viceversa.

La classe Buyer ha tra i suoi attributi una stringa nome univoca all'interno del seller e una lista di Request_B effettuate (a quel particolare seller); saranno inoltre presenti i metodi getter e setter, il *toString* e un metodo *addRequest* per aggiungere una richiesta alla lista; mentre il nome lo si può definire direttamente nel costruttore.

La classe Seller ha una stringa univoca (all'interno del buyer) che ne rappresenta il nome; una lista di richieste ricevute Request (da quel Buyer) e una lista di offerte fatte Offer_S (a quel buyer); inoltre ci sono i metodi getter e setter, il *toString* ridefinito e i metodi *addRequest* e *addOffer* per aggiungere gli elementi nelle due liste; mentre il nome lo si può definire direttamente nel costruttore.

La classe Request deve contenere il nome del buyer, una descrizione della richiesta, una lista di Seller a cui va inviata la richieste una lista di Offer_S che rappresenta le offerte ricevute dai seller designati per quella richiesta. Conterrà i metodi getter e setter, il *toString*, i metodi per aggiungere gli elementi nella lista, tra cui *l'addSeller* di cui abbiamo parlato prima; mentre il nome e la descrizione si possono definire direttamente nel costruttore.

La classe Request_B deve contenere un riferimento ad un Buyer, ad un offerta Offer e la descrizione della richiesta. Serviranno i metodi getter e setter, il *toString* ridefinito; mentre la descrizione la si può definire direttamente nel costruttore.

La classe Offer deve contenere un riferimento alla richiesta Request_B, un intero che rappresenta la somma proposta e uno stato in cui verrà scritto l'esito della decisione del buyer. Anche qui ci saranno i metodi getter e setter, il *toString* ridefinito e la somma proposta verrà generata nel costruttore sfruttando la classe Random e 2 valori interi che indicano gli estremi dell'intervallo dalla quale verrà pescato il numero casuale.

Infine la classe Offer_S deve contenere un riferimento alla richiesta Request, al Seller, un intero che rappresenta la somma proposta dal seller e una stringa che rappresenta lo stato dell'offerta. Conterrà i metodi getter e setter, il *toString* ridefinito e la somma viene definita nel costruttore. Notare che la definizione di queste classi è risultata semplice proprio grazie alla definizione del modello informativo.

In seguito l'utente dovrà definire le classi che implementano i vari task del sistema; definendo non solo guardie e azioni, ma anche i metodi tipici dell'interazione B2B nelle transizioni di tipo send e

receive, ossia i metodi di TCode *getPayload*, *genEntity* o *updateEntity*, *getRecipients* e *getCorrelationKey*.

La prima transizione è Requests: essa non ha né guardia né azione, ma ridefinisce il metodo *getPayload* che restituisce la descrizione della richiesta; ridefinisce *getRecipients* che restituisce la lista dei seller associati alla richiesta; ridefinisce getCorrelationKey che restituisce la chiave (ID) dell'oggetto Request associato; infine ridefinisce genEntity che genererà una Request_B nella rete del seller, inizializzata con le informazioni contenute nel messaggio (stringhe) e ricavando gli oggetti correlati grazie al metodo di Net *getEntityByType* che restituisce la lista di tutte le entità della rete del tipo specificato. Inoltre, essendo un task generativo, va ridefinito *isGenTask* in modo che ritorni true.

La transizione GeneratesOffer non è un task B2B, quindi non ridefinirà tali metodi; non ha nessuna guardia, ma ha un'azione: essa crea un token Offer e utilizza le informazioni contenute nel token di tipo Request_B per collegare (attraverso il settaggio degli attributi) l'offerta a tale richiesta.

Offers è un altro task che implementa un'interazione generativa B2B, pertanto ridefinirà gli stessi metodi ridefiniti dal task Request: nel payload verrà messa la somma proposta; nella lista dei riceventi verrà messo solamente il buyer della richiesta a cui l'offerta si riferisce e come chiave di correlazione verrà utilizzata la chiave della richiesta inviata dal buyer, che è stata memorizzata nel campo extKey dell'oggetto Request_B. Infine verrà ridefinito il metodo *genEntity* che genererà un oggetto Offer_S i cui attributi saranno ricavati dal messaggio inviato (parametri di *genEntity*), mentre la richiesta sarà ricavata grazie alla chiave esterna, passata come parametro al metodo *getEntityByKey* di Net che restituisce l'oggetto data la chiave.

La transizione Evaluates, non è di tipo B2B, ma contiene sia una guardia sia un'azione: la guardia dovrà bloccare la transizione fino a che non saranno presenti almeno 3 oggetti Offer_S nella lista delle offerte dell'oggetto Request associato al token Offer_S di ingresso; l'azione invece dovrà ordinare la lista con tali offerte per prezzo crescente, accettare la prima offerta della lista e rifiutare tutte le altre.

Infine la transizione OffersEval implementa un task B2B di tipo modificativo, quindi ridefinirà *isModTask* in modo che ritorni true; inoltre ridefinirà il metodo getPayload che restituirà lo stato dell'offerta sotto forma di stringa (ossia l'esito della decisione del buyer); il metodo *getRecipients* che restituirà il seller che aveva inviato tale offerta; il metodo *getCorrelationKey* che restituirà la chiave esterna dell'offerta, ossia la chiave che identifica l'oggetto Offer nel sistema informativo di quel seller; Infine verrà ridefinito il metodo *updateEntity*, tipico dell'interazione modificativa, che, usando la chiave esterna, recupererà l'oggetto Offer e né aggiornerà lo stato. Il task non ha né

guardia né azione, e sarà l'ultimo evento prima della fine della simulazione (nessun task può scoccare non essendoci tokens in circolo nella rete).

9.1.5 Risultati ottenuti

Questo esempio semplice di processo B2B mirava a testare il funzionamento dell'ambiente di simulazione soprattutto per quanto riguarda le interazioni B2B (la simulazione della rete interna l'abbiamo vista nello scorso capitolo): le interazioni generative devono creare un oggetto a partire dalle informazioni ricevute nel payload del messaggio e spesso questo nuovo oggetto contiene dei riferimenti ad altri oggetti del proprio sistema informativo o di quello di un partner; quelle modificative invece devono recuperare un oggetto del proprio sistema informativo e modificarne un valore. Per questo motivo abbiamo inserito nell'entità 2 campi: il primo rappresenta una chiave interna, il secondo una chiave esterna; in questo è possibile fare riferimento ad un oggetto del proprio sistema informativo (chiave interna) oppure ad uno del sistema informativo del partner (chiave esterna).

Fatta questa premessa, i nostri test si sono concentrati proprio sulla parte B2B del processo, e l'ambiente è stato sviluppato per poter simulare senza errori tutti i casi possibili, pertanto sono stati testati casi che potevano portare ad ambiguità nei risultati. Il primo caso testato è quello con 1 buyer e 3 seller in cui il buyer effettua una sola richiesta (quindi 1 solo token viene inserito in "r1"); inoltre, la somma proponibile da ogni seller può avere un valore minimo di 10 e un valore massimo di 30. Simulando questo primo caso, vediamo che il buyer accetta l'offerta del primo seller con importo 11 e rifiuta le altre due con importi 21 e 29 (essendo i valori generati in maniera casuale, la ripetibilità non è garantita). La stampa a fine simulazione mostra tutte le entità di tutte le reti; da questa possiamo osservare che i valori siano effettivamente corretti e notare eventuali errori nei protocolli di collaborazione: in questo caso sia l'Offer_S del buyer, sia l'Offer corrispondente del seller hanno lo stesso stato e lo stesso importo, quindi la correlazione è avvenuta con successo attraverso l'invio di chiavi di correlazione.

Nel secondo test, proviamo a complicare la situazione, aggiungendo un buyer: questa volta, oltre ad accettare e rifiutare le offerte,bisogna anche distinguere le richieste provenienti da i diversi buyer e inviare l'offerta al buyer corretto. Anche in questo caso la simulazione funziona correttamente, riuscendo a distinguere le richieste dei vari buyer ed aggiornando il proprio sistema informativo nella maniera corretta.

L’ultimo test complica ulteriormente la situazione: vengono inseriti 2 token in “r1” nel primo buyer e 1 token in “r1” nel secondo buyer: il primo buyer quindi, effettua due richieste contemporaneamente. La difficoltà sta nella transizione Evaluates che deve attendere 3 offerte riferite alla stessa richiesta, prima di poterle valutare; questo problema viene risolto grazie alla chiave di correlazione inviata dal buyer attraverso Requests e ricevuta in seguito nel messaggio inviato tramite la transizione Offers del seller; permettendo l’associazione corretta. Anche questa volta, la stampa dei risultati dimostra la correttezza delle varie associazioni poiché le richieste e le offerte associate hanno uguali valori in entrambi i sistemi informativi.

Non essendoci concentrati sulla caratteristica temporale del processo, non abbiamo analizzato le reti in cerca di bottleneck o di possibili miglioramenti, ma ci sia concentrati sugli aspetti funzionali del sistema. Tuttavia, utilizzando l’ambiente di simulazione e definendo con cura le classi del sistema, è possibile simulare qualunque processo di business, facendo attenzione al contenuto dei messaggi inviati ai partners: essi devono contenere solo le informazioni strettamente necessarie allo svolgimento del processo e alla eventuale correlazione con oggetti del proprio sistema informativo. Inserendo le tempistiche è possibile fare analisi tipiche dei processi di business (come quelle fatte nei precedenti capitoli), ma in questo caso, essendo un processo B2B, viene testata anche l’effettiva funzionalità dei protocolli di collaborazione stabiliti ed eventualmente vengono rilevate delle imperfezioni (ad esempio errori dovuti alla mancata presenza delle chiavi).

Capitolo 10

Conclusione

10.1 Possibili sviluppi futuri

L'ambiente di simulazione per processi B2B che abbiamo sviluppato può essere applicato a tutti i processi che possono essere rappresentati da una o più reti di Petri; ma la sua applicazione non è banale: l'utente deve conoscere il linguaggio Java, oltre ad avere la documentazione con l'elenco delle funzioni di libreria, i parametri e i valori di ritorno. Pertanto l'utente tipico è un addetto che conosce almeno le basi del linguaggio Java, in modo da poter scrivere le classi che estenderanno il concetto di risorsa (Entity) e di task (TCode).

Per estendere l'utilizzo dell'ambiente anche a utenti più generici si potrebbe sviluppare un'interfaccia grafica:

- Si potrebbe costruire la rete attraverso una grafica 2D, inserendo posti, transizioni e collegamenti.
- Anche il modello informativo si potrebbe costruire sfruttando tale interfaccia grafica, specificando non solo le entità in gioco, ma anche gli attributi che le caratterizzano.
- Di conseguenza si può definire anche il modello collaborativo, verificando che le entità coinvolte rispettino il sistema informativo.
- Si potrebbero definire le classi che estendono Entity (i tokens) fandando a riempire dei template con le informazioni specifiche del tipo di token.
- Analogalmente, anche i task possono essere definiti sfruttando i template e delle funzioni atte a riempirli.
- I risultati stampati a video, potrebbero essere visualizzati nell'interfaccia in maniera più accurata; in generale tutte le fasi potrebbero essere visualizzate e addirittura si potrebbe creare un modello 3D in cui osservare passo passo come agisc eil sistema simulato.

Naturalmente, questa interfaccia richiede lo sviluppo di tutte le funzioni che implementano la funzionalità della GUI (Graphic User Inteface): il concetto è molto simile a quello delle interfacce grafiche dei software di simulazione allo stato dell'arte, la cui interfaccia è user-friendly ed è

particolarmente accurata; semplificando l'utilizzo del software ed aumentando l'impatto dei risultati sui colleghi che spesso non sono esperti di sistemi di simulazione.

Inoltre, sarebbe molto utile inserire un meccanismo in grado di stoppare, rallentare e velocizzare la simulazione: l'esecuzione con stampa finale è molto rapida; questi tasti potrebbero favorire il debugging dei moduli definiti dall'utente e migliorare la resa visiva della simulazione.

Un altro possibile sviluppo riguarda il modo di fornire i dati di input al simulatore: si potrebbe ad esempio sfruttare il linguaggio XML, definendo uno schema che definisce la struttura e il contenuto degli oggetti, in tal modo sarebbe anche possibile sviluppare il simulatore sotto forma di applicazione web che simula un sistema in base ai dati XML forniti e alle funzioni utilizzate dall'interfaccia del client. L'alternativa a XML è usare fogli di calcolo Excel con id ati necessari a creare le varie componenti del sistema, e quindi serve un serializzatore che riceve tali file e costruisce le classi associate.

Infine si potrebbe utilizzare un database per immagazzinare i dati in un certo formato ed eventualmente analizzarli in seguito, quindi implementare questo meccanismo attraverso Hibernate. ossia uno strato di middleware che consente allo sviluppatore di automatizzare le procedure per le operazioni cosiddette CRUD (Create, Read, Update, Delete) dei database.

Bibliografia

Gli esempi sono stati tratti dalle slides del corso di Ingegneria del software, 05BIDOV, Politecnico di Torino, a.a. 2015/2016;

L'esempio Production System 1 è stato tratto dalle slides del corso di Ingegneria del software, 05BIDOV, Politecnico di Torino, a.a. 2015/2016;

https://en.wikipedia.org/wiki/Process_modeling

Slides del corso di Ingegneria del software, 05BIDOV, Politecnico di Torino, a.a. 2015/2016;

https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html

https://en.wikipedia.org/wiki/Petri_net

http://www.bpsim.org/

http://www.bpsim.org/specifications/2.0/WFMC-BPSWG-2016-01.pdf

https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation

https://www.flexsim.com/

https://www.arenasimulation.com/

https://en.wikipedia.org/wiki/Discrete_event_simulation

https://en.wikipedia.org/wiki/List_of_discrete_event_simulation_software

http://jaamsim.com/

https://www.simul8.com/discrete-event-simulation

https://www.cs.cmu.edu/~music/cmsip/readings/intro-discrete-event-sim.html