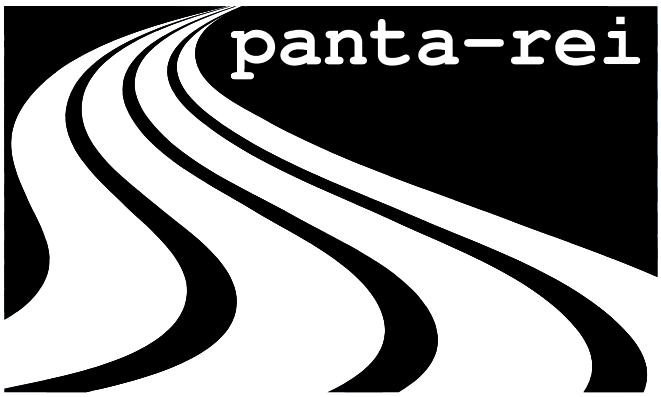
Università degli studi di Pisa
Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Informatica

Sistemi in tempo reale



Panta Rei

Real time scheduling simulator

a cura di
Sacco Cosimo

Indice

Introduzione **v**
0.1 Obiettivo v
0.2 Struttura dell' elaborato v
0.3 *Panta Rei*: tutto scorre v

1 Requisiti **1**
1.1 Requisiti funzionali 1
1.2 Requisiti non funzionali 2

2 Progetto **3**
2.1 Model 4
2.1.1 Timer 5
2.1.2 Task 5
2.1.3 System Queues 7
2.1.4 Activator 8
2.1.5 Scheduler 8
2.1.6 Interazioni 17
2.2 View 18
2.2.1 GnuplotSchedulingEventVisitor 19
2.3 Controller 19
2.3.1 CommandInterpreter 19
2.3.2 PantaReiLanguage 20

A Librerie **23**
A.1 Librerie interne 23
A.1.1 CommandInterpreter 23
A.1.2 DesignPatterns 24
A.1.3 Queue 26
A.1.4 EventManagement 26
A.2 Librerie esterne 27
A.2.1 GnuplotCpp 27
A.2.2 Boost Program Options 27
A.2.3 Boost Static Assert e Boost Type Traits 27

A.2.4 Boost Spirit 27

Introduzione

0.1 Obiettivo

L' elaborato si propone di descrivere il simulatore di scheduling real-time *Panta Rei*, sviluppato nell' ambito del corso di *Sistemi in Tempo Reale*.

0.2 Struttura dell' elaborato

La relazione si compone di due capitoli e un' appendice. Il capitolo 1 riporta la *mission* dell' applicazione e ne espone i requisiti funzionali e non funzionali. Nel capitolo 2 sono sviluppati gli aspetti riguardanti la progettazione architetturale e le soluzioni dei problemi principali. L' appendice A, infine, riporta un elenco di librerie, interne ed esterne al progetto, utilizzate in fase di implementazione.

0.3 *Panta Rei*: tutto scorre

> $\pi\acute{\alpha}\nu\tau\alpha\ \rho\varepsilon\tilde{\imath}\ \grave{\omega}\varsigma\ \pi\sigma\tau\alpha\mu\acute{\sigma}\varsigma$
> *Tutto scorre come un fiume.*

Non è possibile bagnarsi due volte nello stesso fiume:

- nessun *istante* è uguale ad un altro;
- tutto *muta dinamicamente*;

Per questo ho scelto di chiamare il simulatore di scheduling real-time *Panta Rei*.

Capitolo 1

Requisiti

Vengono esposti, qui di seguito, i requisiti che *Panta Rei* deve soddisfare, suddivisi in *requisiti funzionali*[1] e *requisiti non funzionali* [2].

1.1 Requisiti funzionali

Panta Rei deve consentire di simulare l' andamento di una *schedulazione real-time* a partire da una situazione iniziale configurabile secondo i seguenti gradi di libertà:

- *durata*: deve essere possibile la specifica della durata massima della simulazione;
- *task*:
  - *numero*: l' utente deve poter scegliere il numero di task coinvolti nella simulazione;
  - *parametri*:
    - *arrival time* (a): non devono essere posti vincoli sull' istante di arrivo dei task [3] ;
    - *computation time* (C): non devono essere posti vincoli sulla durata del tempo di computazione dei task [3] ;
    - *relative deadline* (D): non devono essere posti vincoli sulla durata della deadline relativa [3] [4];
    - *period* (T)[5]: non devono essere posti vincoli sulla durata del periodo [3] [6];

[1] Requisiti di dominio, relativi alle funzionalità specifiche dell' applicazione.
[2] Requisiti non direttamente collegati alle funzionalità di dominio.
[3] Oltre, naturalmente, al vincolo di non negatività.
[4] Oltre, naturalmente, al vincolo di coerenza $D \geq C$.
[5] Ove tale grandezza sia significativa.
[6] Oltre, naturalmente, al vincolo di coerenza $T \geq D$.

- *algoritmo di scheduling*:
  - *preemptiveness*: deve essere possibile specificare se l' algoritmo scelto debba comportarsi in modo *preemptive*;
  - *policy*: deve essere possibile specificare la *politica di scheduling*, offrendo:
    - *Rate Monotonic*;
    - *Deadline Monotonic*;
    - *Earliest Deadline First*;

1.2 Requisiti non funzionali

Panta Rei deve offrire all' utente le seguenti funzionalità:

- *modalità interattiva*: all' utente deve essere permessa l' interazione con l' applicazione secondo un' interfaccia interattiva, *CLI* oppure *GUI*;
- *modalità batch*: l' applicazione deve permettere l' esecuzione non interattiva, definendo un opportuno *linguaggio di script*;
- *visualizzazione grafica*: l' applicazione deve consentire all' utente di visualizzare il grafico relativo alla schedulazione simulata;
- *logging*: l' applicazione deve permettere di impostare due *file di log*:
  - *log file*: file di log che registrerà in formato testuale l' andamento della simulazione;
  - *error log file*: file di log che conterrà eventuali messaggi di errore lanciati dall' applicazione;

Capitolo 2

Progetto

Il design di *Panta Rei* è stato sviluppato per permettere una semplice riadattabilità delle sue parti. Per ottenere questo risultato sono state utilizzate tecniche di progettazione *component based* e *object oriented*.

La macroarchitettura di *Panta Rei* si basa sul *design pattern Model-View-Controller*: l'intera applicazione è pensata come aggregazione e interazioni fra tre componenti:

- *Model*: il componente che ospita le strutture dati e la logica dell' applicazione;
- *View*: il componente che permette la visualizzazione;
- *Controller*: il componente che permette l' interazione con l' utente;

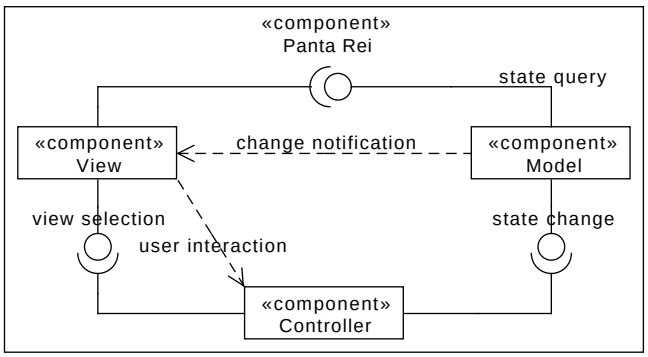


Figura 2.1: Componenti Model, View, Controller

Nella figura 2.1 sono evidenziate le relazioni che legano i componenti tra loro. Le interfacce esposte dal componente *Model*, *state query* e *state change*, consentono, rispettivamente, di consultare lo stato attuale delle strutture

dati interne e di modificarlo. Il componente *View* permette, attraverso l' interfaccia *view selection*, di modificare le modalità di visualizzazione dell' applicazione[1]. Il componente *Controller*, infine, mantiene la logica di controllo dell' applicazione.

Il componente *Model* può notificare, sfruttando un meccanismo di *event handling*, un cambiamento di stato al componente *View*, mentre quest' ultimo può fornire, attraverso lo stesso meccanismo, delle direttive al componente *Controller* circa le azioni dell' utente[2]. Vengono presentati, di seguito, i singoli componenti di *Panta Rei*.

2.1 Model

In figura 2.2 vengono illustrate le parti che costituiscono il componente *Model* e le principali relazioni che le legano. Il design del componente si

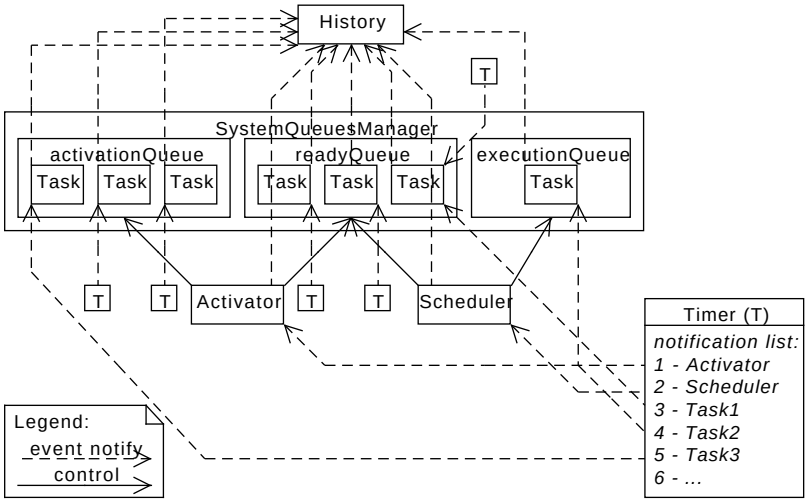


Figura 2.2: Organizzazione del componente *Model*

basa sul semplice meccanismo di *event handling* offerto dal *design pattern Observer*(vedi A.1.2.1). Venogo analizzate, di seguito, le singole parti che costituiscono il componente *Model*.

[1]Attualmente questa interfaccia ammette una sola modalità di visualizzazione.

[2]Attualmente l' interazione, che è limitata alla *CLI*, viene gestita direttamente dal componente di controllo, mentre la visualizzazione sfrutta il più tradizionale meccanismo di query attiva verso il componente *Model*.

2.1.1 Timer

Il `Timer` è un puro produttore di eventi temporali. Come si può vedere

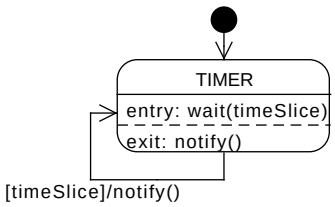


Figura 2.3: Statechart del componente *Timer*

dallo statechart in figura 2.3, il `Timer` notifica il passaggio di ogni *time slice* [3] a tutti i componenti registrati (vedi A.1.2.1). L' ordine di notifica dell' evento temporale è ben definito, ed è il seguente:

1. `Activator`;
2. `Scheduler`;
3. `Task1`;
4. ...

Tale ordine riflette la *consecutio actionis* dei diversi componenti: al passaggio di ogni *time slice*, innanzitutto, l' `Activator` attiva (sposta in coda pronti) i task per i quali è giunto il tempo d' attivazione, dopodiché lo `Scheduler` può schedulare il task da eseguire, infine ciascun `Task` può essere aggiornato per la prossima *time-slice*.

2.1.2 Task

Un `Task`, in *Panta Rei*, costituisce l' unità minima di schedulazione, e mantiene uno stato temporale[4] che viene aggiornato alla ricezione di ogni evento temporale. La legge di aggiornamento dello stato temporale (in seguito: *microstato*) varia a seconda dello stato di esecuzione (in seguito: *macrostato*) nel quale il task si trova. Ai fini della semplice simulazione, *Panta Rei* prevede i seguenti *macrostati*:

[3] *Panta Rei* è un simulatore *off-line*, pertanto il Timer attende la scadenza di una *time slice* di durata 0.

[4] Ovvero, un insieme di variabili riferite all' evoluzione temporale del task, come ad esempio il numero di *time slice* di esecuzione mancanti, il numero di *time slice* passate dall' arrivo della *deadline* e così via.

- `NEW`: il task è in attesa del proprio istante di attivazione;
- `READY`: il task è in attesa di essere schedulato;
- `EXECUTING`: il task è in esecuzione;

mentre le *variabili di microstato sono*:

```
/* Per task periodici e aperiodici */
const unsigned int arrivalTime;
const unsigned int computationTime;
const unsigned int relativeDeadline;
unsigned int absoluteDeadline;
unsigned int elapsedTime; /* Dall' attivazione */
unsigned int remainingComputationTime;
unsigned int instantaneousExceedingTime;
unsigned int pendingInstances;
/* Solo per task periodici: */
const unsigned int period;
unsigned int currentInstanceArrivalTime;
int remainingPeriod;
```

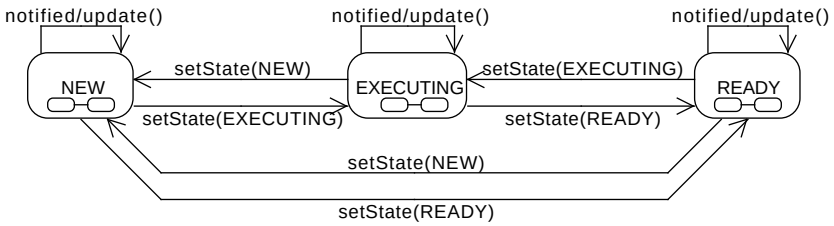


Figura 2.4: Statechart di un *Task*

Come si può vedere in figura 2.4, l' aggiornamento del *microstato* avviene in conseguenza della notifica di un evento temporale, mentre le transizioni di *macrostato* vengono attivate dalla chiamata esplicita al metodo `setState`; pertanto, una transizione autonoma di *macrostato* non è possibile. Tale scelta è coerente con la *mission* esposta all' inizio del capitolo 2: infatti, una transizione di *macrostato* è necessariamente legata alla *legge di schedulazione*, laddove le transizioni di *microstato* sono dipendenti semplicemente dalla *legge di aggiornamento del microstato*, che varia seguendo il *macrostato* attuale, mantenendosi indipendente dalla *legge di schedulazione*. Supponiamo, per esempio, che un `Task` si trovi nel *macrostato* `EXECUTING`: la ricezione di un evento temporale potrebbe comportare la transizione verso un *microstato* nel quale il valore booleano indicante *deadline miss* è `true`, e questo è

indipendente dalla particolare *legge di schedulazione*, mentre la revoca del processore, ad esempio, a causa di *preemption*, e la conseguente transizione verso il *macrostato* `READY`, è legata alla *legge di schedulazione*. Pertanto, volendo garantire la possibilità di variare la *legge di schedulazione* senza modificare le *leggi di aggiornamento del microstato*, risulta necessario disaccoppiare le due leggi in modo che la *legge di schedulazione* sia gestita da un modulo esterno al `Task`. L' alternativa sarebbe stata l' *hard coding* della *legge di schedulazione* all' interno dei `Task`, il che avrebbe implicato una riscrittura delle *leggi di schedulazione* in forma distribuita, obiettivo molto più complesso, avrebbe reso la soluzione meno elegante e, soprattutto, avrebbe sottratto dinamismo e riadattabilità.

2.1.3 System Queues

In *Panta Rei* vengono simulate le seguenti code di sistema:

- `activationQueue`: coda di attivazione;
- `readyQueue`: coda dei task pronti;
- `executionQueue`: coda di esecuzione[5];

Il `SystemQueuesManager` offre un servizio di *naming* delle code di sistema, che risultano così accessibili da tutti gli altri componenti. Ciò che viene reso accessibile, in realtà, è una semplice interfaccia, la quale riferisce un oggeto implementazione che definisce opportunamente la struttura dati e i metodi di accesso alla stessa (vedi A.1.3). Il disaccoppiamento tra interfaccia di una coda (pubblicamente accessibile attraverso il servizio di *naming*) e la sua implementazione consente di ottimizzare le politiche di accesso agli elementi: `Scheduler` diversi potrebbero necessitare di code organizzate in maniera diversa, e di ciò essi soli ne sono al corrente; è opportuno, quindi, permettere all' owner di una coda di definire la sua implementazione, mentre l' accesso è normalmente garantito ai moduli richiedenti attraverso degli oggetti interfaccia. A mo' d' esempio: uno `Scheduler` implementante la politica *Rate Monotonic* abbisogna di una coda dei task pronti ordinata per periodo crescente; ma anche l' `Activator` ha bisogno di accedere a questa coda: allora, lo `Scheduler` richiederà al `SystemQueuesManager` l' interfaccia dal nome `readyQueue`, e inizializzerà il suo riferimento all' oggetto implementazione con un oggetto del tipo opportuno. In questo modo la coda viene condivisa in maniera sicura e l' owner può stabilirne dinamicamente l' implementazione.

[5] *Panta Rei* simula algoritmi di scheduling monoprocessore, pertanto questa coda avrà un solo elemento.

2.1.4 Activator

L' `Activator` è il componente incaricato di attivare i `Task`, ovvero, del loro ingresso nella coda dei task pronti. L' `Activator` fornisce l' implementazione della `activationQueue`, si tratta di una coda ordinata secondo l' istante di attivazione dei task. Quando viene notificato un evento temporale l' `Activator` procede ad estrarre dalla `activationQueue` tutti i `Task` per i quali è giunto l' istante di attivazione, e li inserisce nella `readyQueue`, dopo aver attivato, per ciascun `Task` da inserire, la transizione di *macrostato* `NEW`→`READY`.

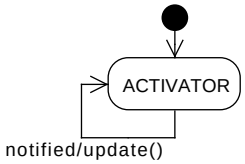


Figura 2.5: Statechart dell' *Activator*

2.1.5 Scheduler

Il componente `Scheduler` provvede a gestire le code `readyQueue` e `executionQueue` secondo la *legge di schedulazione* stabilita. A livello implementativo `Scheduler`

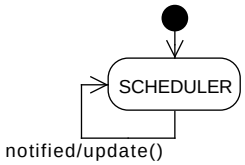


Figura 2.6: Statechart dello *Scheduler*

si presenta come una classe astratta, sono le classi che da essa derivano a definire le specifiche politiche di scheduling. Viene illustrata, ora, l' implementazione di un generico scheduler prioritario.

2.1.5.1 Generic priority scheduling

Algoritmi di scheduling quali *Rate Monotonic*, *Deadline Monotonic* o *Earliest Deadline First*, benché sostanzialmente differenti se comparati sul

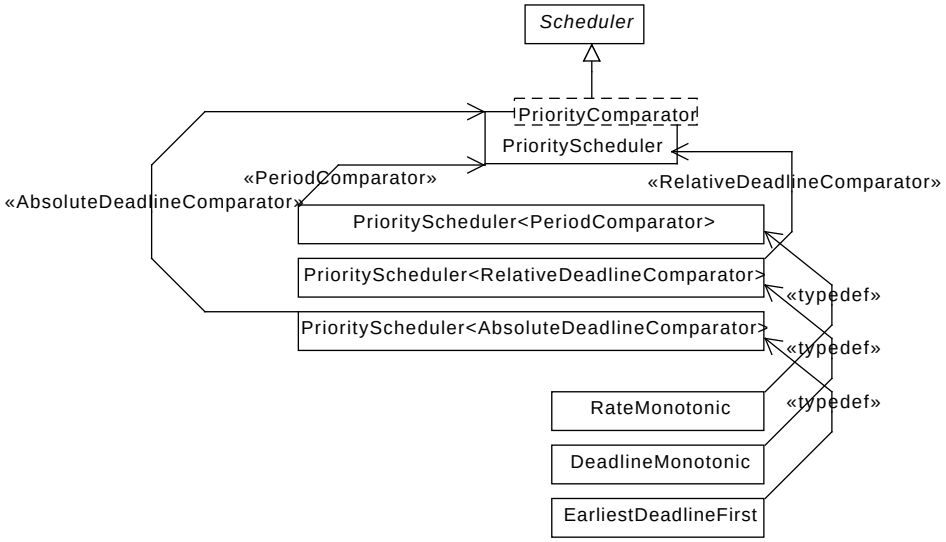


Figura 2.7: Class diagram: generico scheduler prioritario e specializzazioni

piano della schedulazione, ovvero degli *effetti*, condividono la stessa *struttura generica*. Si tratta di algoritmi di scheduling basati su *priorità*; ciò che li differenzia è la particolare definizione di priorità che viene applicata. Sostanzialmente, quindi, un particolare algoritmo di scheduling prioritario può essere visto come un algoritmo prioritario generico *parametrizzato* secondo una certa definizione di priorità, come illustrato in figura 2.7. Pertanto, risulta sufficiente implementare un solo algoritmo generico: per ottenere tutti i possibili algoritmi di scheduling prioritario è sufficiente parametrizzare l' algoritmo generico con le diverse *leggi di priorità*.

L' algoritmo prioritario generico è stato implementato attraverso la tecnica dei *decision tree*.

Decision tree algorithm Viene ora brevemente illustrata questa tecnica, utilizzata per implementare l' algoritmo di scheduling prioritatio generico.

Il primo step di ogni esecuzione di un algoritmo basato su *decision tree* consiste nel ricalcolare il valore di un *vettore di stato booleano*:

$$s = \begin{bmatrix} s_0 \\ \vdots \\ s_n \end{bmatrix}$$

Tale vettore, ottenuto secondo una certa *legge di aggiornamento dello stato*, contiene le coordinate di navigazione all' interno di un *binary tree* i cui rami sono etichettati, a coppie, da *espressioni booleane complementari* relative, di volta in volta, allo stato di una singola *componente di navigazione* (un

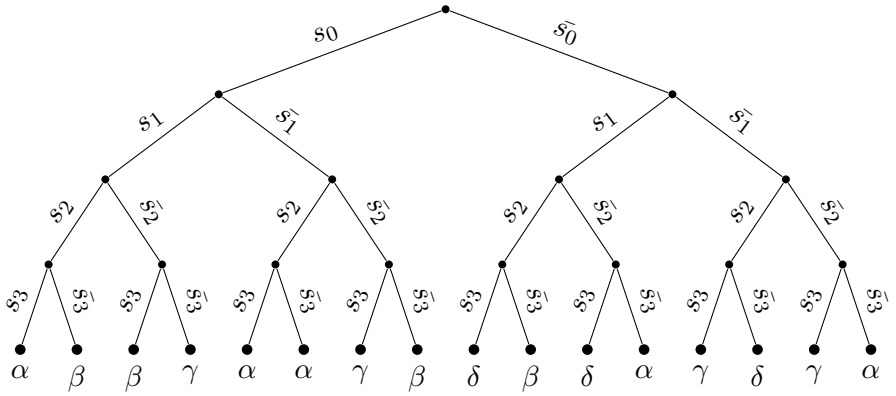


Figura 2.8: Esempio di *decision tree*

esempio in figura 2.8). La navigazione nell' albero procede scegliendo di percorrere, ad ogni bivio, il ramo la cui etichetta risulta verificata. Le foglie dell' albero rappresentano le *decisioni* da prendere. Ad esempio, dato il *vettore di stato*

$$s = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

e il *decision tree* T mostrato in figura 2.8, l' operazione di navigazione $\mathscr{N}$ nell' albero T avrà come risultato

$$\mathscr{N}_T(s) = \gamma$$

Ovvero, la navigazione all' interno del *decision tree* secondo le coordinate $[1, 0, 0, 1]$ porta ad una foglia associata alla decisione γ, che viene dunque eseguita. Si noti come sia possibile ottenere, nell' esempio, la stessa decisione anche partendo da *vettori di stato* differenti.

L' approccio basato su *decision tree* può essere schematizzato, utilizzando il paradigma di programmazione *object oriented*[6], come segue:

```
class DecisionTreeAlgorithm
{
    private:
        DecisionTree t;
        State s;
        void alpha(){/* Decisione alpha */}
        void beta(){/* Decisione beta */}
        .    .    .
        .    .    .
        .    .    .
        void updateState()
```

[6]Si ricorre, a tal proposito, alla sintassi *C++*.

```
        {
            /* Aggiornamento dello stato */
        }
        void navigateDecisionTree()
        {
            /* Navigazione nel decision tree */
            /* Esecuzione della decisione individuata */
        }
    public:
        DecisionTreeAlgorithm(/* ... */)
        {
            /* Inizializzazione di t e s */
        }
        void execute()
        {
            updateState()
            navigateDecisionTree();
        }
};
```

La navigazione all' interno del *decision tree* ha una complessità, se F è il numero di foglie, pari a $\mathscr{O}(log_2(F))$ nell' ipotesi di albero completo e perfettamente bilanciato. Tipicamente, la navigazione all' interno di un albero avviene ricorsivamente. Se $log_2(F)$ è grande, tale approccio può risultare inefficiente. Un modo semplice per rendere la navigazione iterativa è l' innestamento di costrutti `if else` concordemente con la topologia dell' albero. Viene esposta, di seguito, una terza via per evitare la navigazione ricorsiva dell' albero.

Sia $\mathscr{D}$ l' insieme delle decisioni, e sia $\mathbb{B}^n$ l' insieme dei possibili vettori di stato[7], possiamo definire i seguenti insiemi:

$$\mathscr{S}_\delta \subseteq \mathbb{B}^n : \\ \mathscr{N}_T(\sigma) = \delta \ \forall \sigma \in \mathscr{S}_\delta \\ \wedge \\ \nexists \sigma^* \in \{\mathbb{B}^n - \mathscr{S}_\delta\} : \mathscr{N}_T(\sigma^*) = \delta$$

ovvero, per ogni possibile decisione $\delta \in \mathscr{D}$ è definito con $\mathscr{S}_\delta$ l' insieme di tutti e soli gli stati tali per cui $\mathscr{N}_T(\sigma) = \delta,\ \sigma \in \mathscr{S}_\delta$. È facile notare che, fissata una decisione δ, possiamo associare ad ogni $s \in \mathscr{S}_\delta$ un unico *mintermine* m che viene verificato proprio da s. Tali mintermini sono tutti e soli i percorsi radice-foglia che conducono alla decisione δ. Nell' esempio precedente (figura

[7] Certamente n è uguale al numero di livelli del percorso radice-foglia più lungo nel *decision tree* e, qualora l' albero sia completo, perfettamente bilanciato e ciascuna foglia sia diversa da tutte le altre, avremo anche che $n = log_2(\sharp\mathscr{D})$.

2.8), alla decisione δ corrisponde l' insieme di stati

$$\mathscr{S}_\delta = \left\{ \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$$

e l' insieme di mintermini

$$\mathscr{M}_\delta = \{\bar{s_0}s_1s_2s_3, \bar{s_0}s_1\bar{s_2}s_3, \bar{s_0}\bar{s_1}s_2\bar{s_3}\}$$

Possiamo affermare che la decisione δ viene presa quando il vettore di stato s rende vera la seguente espressione booleana:

$$f_\delta(s) = \sum_{m \in \mathscr{M}_\delta} m$$

come si può notare, si tratta di una relazione scritta in forma *sum of products*. Dato un *decision tree* T, pertanto, è semplice costruire una famiglia di espressioni booleane

$$\mathscr{F} = \{f_\varphi, \varphi \in \mathscr{D}\}$$

dove

$$f_\varphi(s) = \sum_{m \in \mathscr{M}_\varphi} m$$

e pertanto

$$f_\varphi(s) = \begin{cases} 1 & \mathscr{N}_T(s) = \varphi \\ 0 & \text{altrimenti} \end{cases}$$

Adesso che siamo in possesso di questa famiglia di funzioni, è semplice scrivere una funzione di navigazione non ricorsiva:

```
void navigateDecisionTree()
{
    if(/* f_alpha(s) */)
    {
        alpha();
        return;
    }
    if(/* f_beta(s) */)
    {
        beta();
        return;
    }
    .
    .
    .
}
```

Si può notare come questa implementazione sia, in generale, meno efficiente dell' alternativa basata sulla ricostruzione del *decision tree* innestando costrutti `if else`[8], tuttavia essa risulta più versatile e semplice da utilizzare in fase di programmazione. Come si può vedere, infatti, la compressione della logica di navigazione nella valutazione di poche espressioni booleane consente di appiattire il livello di innestamento dei costrutti `if else` a 1, rendendo il codice molto più leggibile rispetto all' alternativa a $log_2(F)$ livelli di innestamento. Possiamo pensare, inoltre, di semplificare le espressioni booleane ottenute applicando metodi quali l' *algoritmo di Quine-McLuskey* o metodi di ottimizzazione euristica, migliorando ulteriormente la leggibilità del codice.

Implementazione dello scheduler prioritario generico Viene illustrata, ora, l' implementazione dell' algorirmo di scheduling prioritario generico.

Definiamo, innanzitutto, il *vettore di stato* o *navigazione*: si tratta di un vettore $s \in \mathbb{B}^6$ con la seguente *legge di aggiornamento*:

$$s = \begin{bmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{bmatrix}^T =$$

$$\begin{bmatrix} \texttt{readyQueue->size() == 0} \\ \texttt{executionQueue->size() == 0} \\ (\bar{s_1})(\texttt{(executionQueue->front())->getRemainingComputationTime() == 0}) \\ (\bar{s_1})(\texttt{(executionQueue->front())->getPendingInstances() == 0}) \\ \texttt{preemptionActivated} \\ (\bar{s_0})(\bar{s_1})(\texttt{comparator(readyQueue->front(), executionQueue->front())}) \end{bmatrix}$$

Possiamo facilmente notare come tale *legge di aggiornamento* definisca il seguente insieme di *stati ammissibili*[9]:

$$\mathscr{S}_A = \left\{ \mathbb{B}^6 - \underbrace{\left\{ \begin{bmatrix} - \\ 1 \\ 1 \\ - \\ - \\ - \end{bmatrix} \cup \begin{bmatrix} - \\ 1 \\ - \\ 1 \\ - \\ - \end{bmatrix} \cup \begin{bmatrix} 1 \\ - \\ - \\ - \\ - \\ 1 \end{bmatrix} \cup \begin{bmatrix} - \\ 1 \\ - \\ - \\ - \\ 1 \end{bmatrix} \right\}}_{\mathscr{S}_N:\ \textit{stati non ammissibili}} \right\}$$

In figura 2.9 viene mostrato il *decision tree* relativo ai vettori di $\mathbb{B}^6$. Tale

[8] Infatti, nel *worst case scenario*, tutte le espressioni f devono essere valutate prima di giungere alla decisione.

[9] Non tutto lo spazio $\mathbb{B}^6$ è ammissibile: la *legge di aggiornamento* appena definita non potrà mai produrre, ad esempio, un *vettore di stato* con componenti s_1 ed s_2 entrambe uguali a 1.

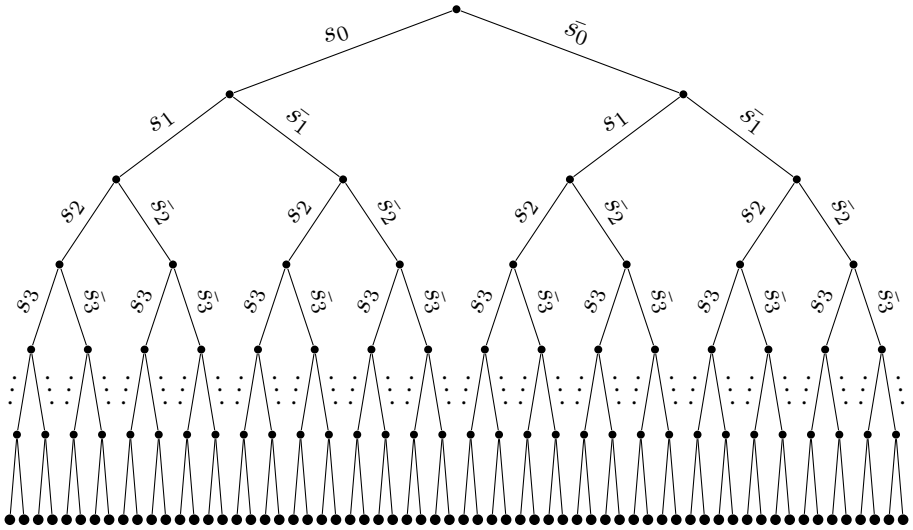


Figura 2.9: *Decision tree* completo relativo a $\mathbb{B}^6$

albero garantisce una varietà di ben 64 decisioni. Tuttavia, esaminando i possibili percorsi e considerando il *significato* di ciascun relativo *mintermine*[10] è semplice constatare che, in realtà, le *decisioni di schedulazione* possibili sono solo 7:

- α: nessuna azione;
- β: rischedula il task in esecuzione;
- γ: riattiva[11] il task in esecuzione;
- δ: schedula un task pronto;
- ε: riattiva il task in esecuzione e schedula un task pronto;
- φ: riattiva il task in esecuzione, inseriscilo in coda pronti e schedula un task pronto;
- ω: inserisci il task in esecuzione nella coda pronti e schedula un task pronto;

[10] Ad esempio, il significato del *mintermine* $\bar{s_0}\bar{s_1}\bar{s_2}\bar{s_3}s_4s_5$ è: *"la coda dei task pronti non è vuota, c' è un task in esecuzione che non ha terminato la computazione e non ci sono, dello stesso task, altre istanze pendenti; inoltre, la preemption è attiva ed esiste un task pronto a priorità maggiore di quella del task in esecuzione"*.

[11] Con la riattivazione lo *Scheduler* cede all' *Activator* il controllo sul task. L' *Activator*, a sua volta, potrà reinserirlo, se il task è periodico, nella *activationQueue*, o terminarlo nel caso di task aperiodico. *Panta Rei*, in questa fase, ammette solamente task periodici.

Si pone

$$\mathscr{N}_T(s) = \dagger \ \forall s \in \mathscr{S}_N$$

ovvero, vengono marcate con † tutte le foglie che sono il risultato della navigazione dell' albero utilizzando *coordinate non ammissibili*. In figura 2.10 viene riportato il *decision tree* completo delle decisioni. Si nota facilmente

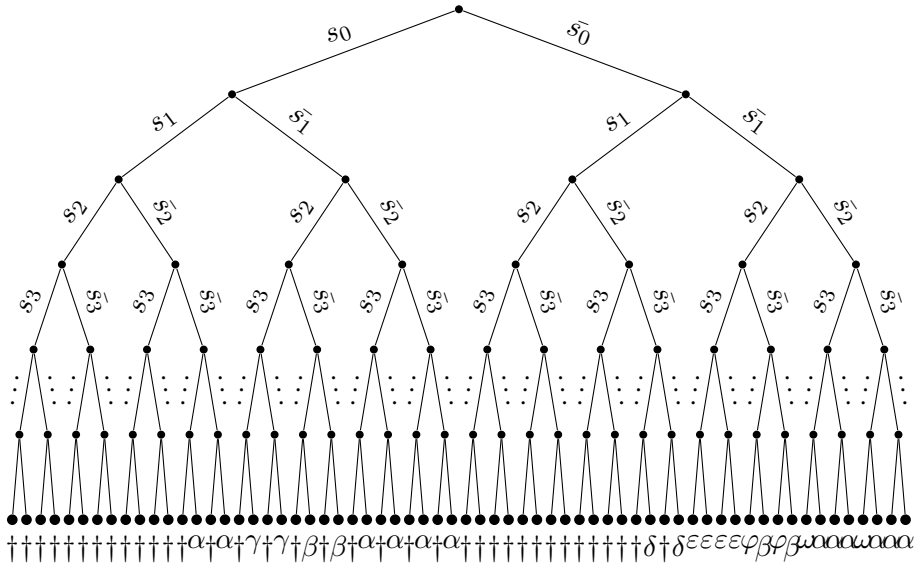


Figura 2.10: *Decision tree* completo per scheduler prioritario

come il numero di *stati non ammissibili* sia considerevole. Inoltre, esistono molte foglie duplicate: ad esempio, tutte le foglie individuate dal *mintermine* $\bar{s_0}\bar{s_1}s_2s_3$ hanno, come etichetta, ε. Possiamo dunque pensare di effetturare, attraverso una procedura di *pruning*, una sostanzale semplificazione dell' albero, scartando tutti gli *stati non ammissibili* e accorpando adeguatamente i percorsi comuni. Tale procedura produce il *decision tree* effettivamente utilizzato, riportato in figura 2.11.

Il *decision tree*, nella sua forma finale, può essere utilizzato, a questo punto, per produrre il codice di un generico scheduler prioritario. La realizzazione delle diverse politiche è demandata semplicemente alla definizione della *relazione di priorità* tra task, in particolare:

- *Rate Monotonic*:

$$\mathscr{P}(t_a, t_b) = \begin{cases} 1 & T_{t_a} < T_{t_b} \\ 0 & \text{altrimenti} \end{cases}$$

- *Deadline Monotonic*:

$$\mathscr{P}(t_a, t_b) = \begin{cases} 1 & D_{t_a} < D_{t_b} \\ 0 & \text{altrimenti} \end{cases}$$

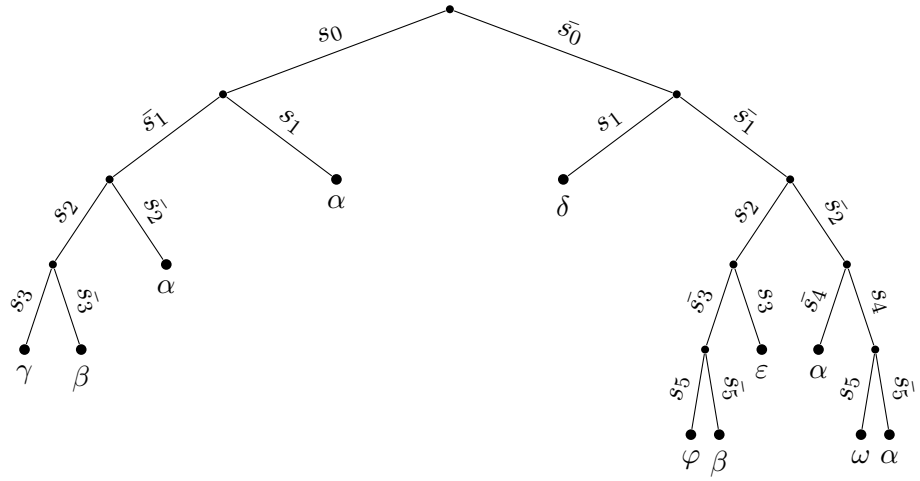


Figura 2.11: *Decision tree* per scheduler prioritario

- *Earliest Deadline First*:

$$\mathscr{P}(t_a, t_b) = \begin{cases} 1 & d_{t_a} < d_{t_b} \\ 0 & \text{altrimenti} \end{cases}$$

dove $\mathscr{P}$ rappresenta la *relazione di priorità*, ed è *vera* se il task t_a ha priorità sul task t_b, mentre T indica il periodo, D la deadline relativa e d quella assoluta.

2.1.5.2 History

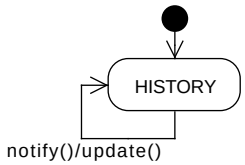


Figura 2.12: Statechart della *History*

L' oggetto `History` rimane in attesa di tutti gli eventi provenienti da `Activator`, `Scheduler` e ciascun `Task`, e provvede a memorizzarli in vista di successive elaborazioni.

2.1.6 Interazioni

In figura 2.2 viene riportata l' organizzazione del componente *Model*, mentre un esempio di esecuzione di uno *step d' automazione* è riportato in figura 2.13. Il `Timer`, producendo eventi temporali e provvedendo alla

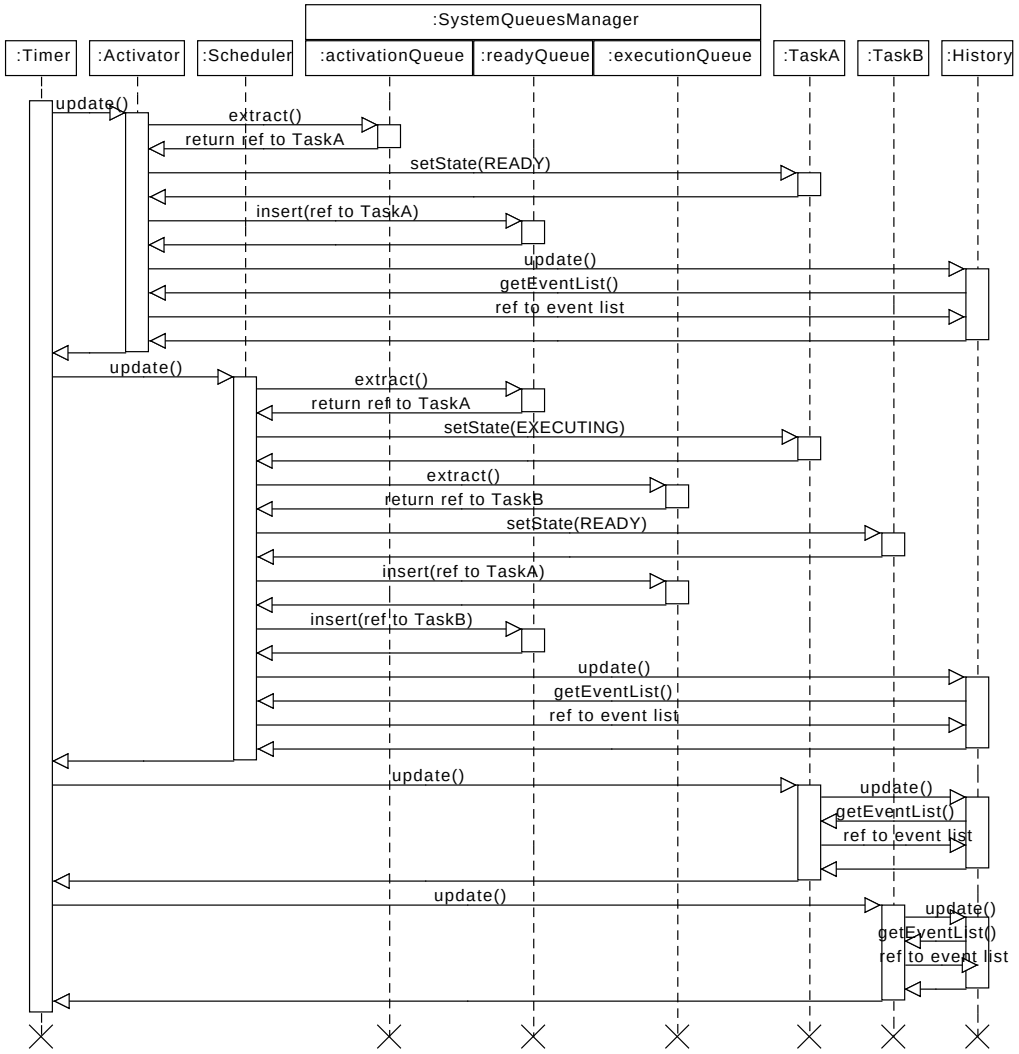


Figura 2.13: Esempio di esecuzione di uno *step d' automazione*

notifica degli stessi a tutti i componenti iscritti, nell' ordine di iscrizione, è il componente che aziona l' intera simulazione. Ogni componente iscritto alle notifiche del `Timer`, all' arrivo di una notifica, procede ad eseguire il relativo *step di automazione*:

- `Activator`: individua, nella `activationQueue`, i `Task` che sono pronti per passare nella `readyQueue`, per ciascuno di essi viene settato l' opportuno *macrostato* (`READY`) e viene trasferito nella `readyQueue`;

- `Scheduler`: in base alla politica di scheduling definita, provvede a spostare i `Task` tra la `executionQueue` e la `readyQueue`, settando l' opportuno *macrostato* (rispettivamente, `EXECUTING` e `READY`), o ridà all' `Activator` il controllo sui `Task` che hanno terminato l' esecuzione;

- `Task`: provvedono ad aggiornare il proprio *microstato* in concordanza col loro attuale *macrostato*;

Ciascuno dei componenti notificati può produrre, eseguendo il proprio *step di automazione*, un certo numero di *eventi di schedulazione*: quando ciò avviene, provvedono a notificarlo al componente `History`, che memorizza l' intera sequenza di eventi nello stesso ordine.

2.2 View

Il modulo *View*, nella versione attuale di *Panta Rei*, offre solamente la possibilità di scegliere la modalità di visualizzazione in risposta ad un esplicito comando dell' utente. In riferimento alla figura 2.1, quindi, è implementato il *concept view selection*, ma non la possibilità di aggiornamento dinamico attraverso meccanismi di *event handling*. La figura 2.14 mostra la composizione del modulo *View*.

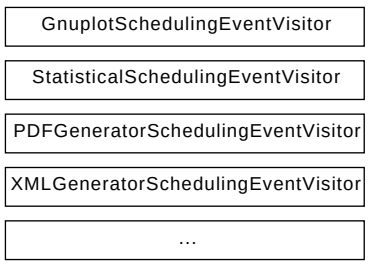


Figura 2.14: Organizzazione del componente *View*

Il componente *View*, nella versione attuale, è una semplice collezione di *visualizzatori*. Ciascun visualizzatore è pensato come un *Visitor* (vedi A.1.2.3) adatto a visitare una collezione di eventi di schedulazione: la visita consente al *visualizzatore* di inizializzare, eventualmente, le strutture dati necessarie alla conseguente visualizzazione. La versione attuale di *Panta Rei* implementa solo il visualizzatore `GnuplotSchedulingEventVisitor`.

2.2.1 GnuplotSchedulingEventVisitor

`GnuplotSchedulingEventVisitor` sfrutta la libreria *GnuplotCpp* (vadi A.2.1) per produrre il grafico annotato della schedulazione. In particolare, questo visitor è in grado di tradurre gli eventi di schedulazione visitati (nell' applicazione, quelli memorizzati in `History` durante la fase di simulazione) in punti relativi a più funzioni: la funzione a gradini prodotta dalla schedulazione, la funzione definita a tratti relativa agli eventi di *deadline miss*, la funzione definita per punti relativa agli eventi di *preemption* e così via. Ciascun punto viene mantenuto in opportune strutture dati (vettori e mappe) che verranno successivamente interpretati dalla libreria *GnuplotCpp* per produrre il grafico.

2.3 Controller

Nella versione attuale di *Panta Rei* l' utente interagisce direttamente col componente *Controller*, attraverso una semplice *CLI*. La figura 2.15 mostra l' organizzazione del componente *Controller*.

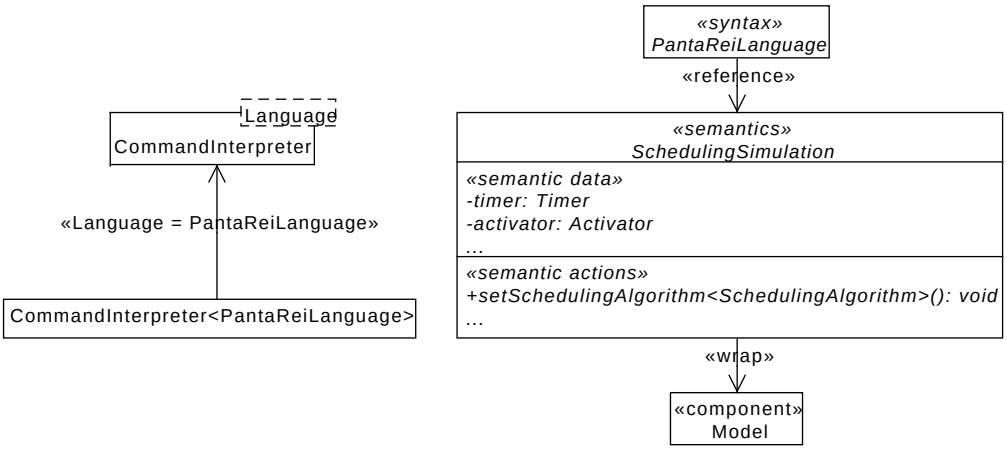


Figura 2.15: Organizzazione del componente *Controller*

2.3.1 CommandInterpreter

Il `CommandInterpreter` è il componente addetto all' interazione con l' utente. Si tratta di un interprete generico, parametrizzabile rispetto al linguaggio `Language`. Un `CommandInterpreter` prevede due modalità di funzionamento: *interattiva* e *batch*. In modalità *interattiva* l' interprete richiede l' inserimento, da parte dell' utente, di statement validi secondo la sintassi

di `Language`, che vengono eseguiti immediatamente, metre in modalità *batch* l' interprete esegue il codice riportato all' interno di uno *script*.

Il linguaggio `Language` deve definire, secondo opportune convenzioni, sia la *sintassi*, in termini di *grammatica*, che la *semantica*, in termini di *azioni* e *strutture dati*[12].

Il componente `CommandInterpreter` è realizzato sfruttando il framework *Boost Spirit* (vedi A.2.4).

2.3.2 PantaReiLanguage

Il *linguaggio di scripting* utilizzato in *Panta Rei* è, nella versione attuale, un semplice *linguaggio regolare*[13]. Esso viene descritto, tuttavia, nel frame più ampio delle *grammatiche libere da contesto*[14] senza limitare, pertanto, la complessità espressiva di future versioni del linguaggo. Segue la descrizione del linguaggio in *Extended Backus Naur Form*[15]:

⟨*script*⟩ ::= ⟨*command*⟩
| ⟨*script*⟩ ⟨*command*⟩

⟨*command*⟩ ::= ⟨*createStatement*⟩
| ⟨*setStatement*⟩
| ⟨*viewStatement*⟩
| '`simulate`'
| '`clear`'
| '`syntax`'
| '`quit`'

⟨*createStatement*⟩ ::= '`create`' ⟨*objectStatement*⟩

⟨*setStatement*⟩ ::= '`set`' ⟨*propertyStatement*⟩

⟨*viewStatement*⟩ ::= '`view`' ⟨*viewer*⟩

⟨*objectStatement*⟩ ::= '`periodic task`' *a C D T*
| ...

[12] Ad esempio, la grammatica del *Java Bytecode* definisce l' istruzione `iadd` (*sintassi*), alla quale è associata l' *azione* "somma i primi due interi sullo stack (*struttura dati*), estraili e lascia il risultato sullo stack".

[13] Grammatica di *tipo 3* nella *gerarchia di Chomsky*, riconoscibile da un *automa a stati finiti*.

[14] Grammatica di *tipo 2* nella *gerarchia di Chomsky*, riconoscibile attraverso un *automa a pila non deterministico*.

[15] Vengono indicate con ... quelle estensioni del linguaggio che derivano in maniera semplice dalle relative estensioni di funzionalità.

⟨*propertyStatement*⟩ ::= '`simulation length`' L
| '`scheduler`' ⟨*schedulingAlgorithm*⟩
| ...

⟨*viewer*⟩ ::= '`Gnuplot`'
| ...

⟨*schedulingAlgorithm*⟩ ::= '`PRM`'
| '`NPRM`'
| '`PDM`'
| '`PDM`'
| '`PEDF`'
| '`NPEDF`'
| ...

Il componente `PantaReiLanguage` è realizzato sfruttando il framework *Boost Spirit* (vedi A.2.4).

Appendice A

Librerie

I problemi incontrati in fase di sviluppo del software sono ascrivibili a due tipologie:

- *problemi di dominio*: si tratta di problemi specifici legati al dominio dell' applicazione, riguardano, principalmente, la *business logic* del progetto;
- *problemi generici*: si tratta di problemi ricorrenti relativi al design e a funzionalità comuni;

Mentre la risoluzione dei *problemi di dominio* viene affrontata in maniera diretta nel progetto del software, può risultare utile slegare la risoluzione dei *problemi generici* dal progetto, scegliendo e producendo delle *librerie* appositamente disegnate.

In questa appendice vengono brevemente presentate le librerie utilizzate nello sviluppo di *Panta Rei*.

A.1 Librerie interne

A.1.1 CommandInterpreter

Un `CommandInterpreter` (figura A.1) è parametrizzato con un `Language`, sul quale è posto il vincolo di derivazione[1] da `boost::spirit::qi::grammar< std::string::const_iterator,boost::spirit::ascii::space_type>`, una delle classi del framework *Boost Spirit* (vedi A.2.4), specializzata rispetto alle `std::string` come *container dei simboli*[2] e `boost::spirit::ascii::space_type` come *skip parser*[3]. Il `CommandInterpreter` riferisce, inoltre, un og-

[1] Tale vincolo può essere imposto grazie alle potenti funionalità delle librerie *Boost Static Assert* e *Boost Type Traits*, vedi A.2.3

[2] Si noti che, in realtà, viene specificato il tipo dell' *iteratore*, non dal *container*.

[3] Ovvero, il *parser* che riconosce una grammatica da "skippare" (in questo caso, la grammatica regolare degli *whitespaces*).

getto di tipo `std::istream`, che rappresenta la *sorgente* del *codice sorgente*, che verrà interpretato. Se `sourceSource == &std::cin`, allora il `CommandInterpreter` funzionerà in modalità interattiva, mostrando un messaggio di benvenuto e un prompt che richiede la digitazione dei comandi. Infine, il metodo `run()` rimane in ascolto di comandi ed esegue il parsing secondo la grammatica definita, utilizzando le funzionalità del framework *Boost Spirit*.

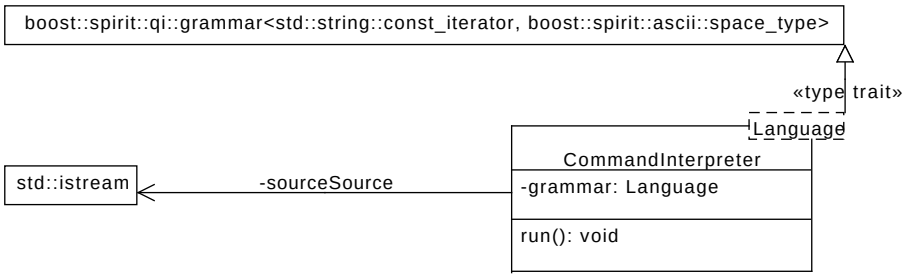


Figura A.1: `CommandInterpreter`

A.1.2 DesignPatterns

Questa libreria raccoglie alcuni utili *design pattern*.

A.1.2.1 Observer

Il *design pattern Observer* consente ad un oggetto di tipo derivato da `Observer` di iscriversi presso un `Subject` affinché questo, quando opportuno, possa notificare all' `Observer` il verificarsi di un evento.

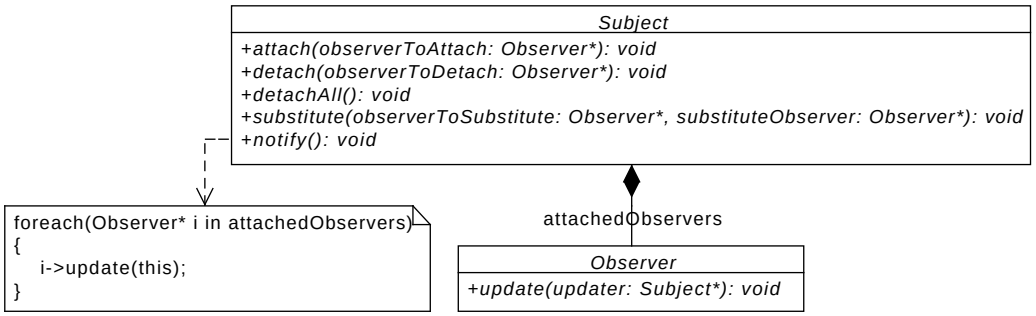


Figura A.2: Design pattern Observer

A.1.2.2 Prototype

Il *design pattern Prototype* espone un metodo di interfaccia, `clone()`, che, opportunamente ridefinito, consente di ottenere un *clone* dell' oggetto invocato.

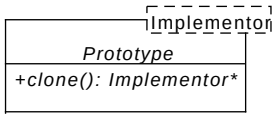


Figura A.3: Design pattern Prototype

A.1.2.3 Visitor

Il *design pattern Visitor* consente di separare le operazioni di navigazione di una collezione dall' algoritmo applicato sugli elementi visitati. Un `VisitorAcceptor` può accettare un `Visitor` e richiedere una *visita* invocando il metodo `visit(...)`, passando il proprio indirizzo come parametro. Il `Visitor`, che attraverso il meccanismo dell' *overloading* ha definito un metodo specifico per ogni specifico `VisitorAcceptor`, provvede ad eseguire la giusta *visita* [4] secondo i propri obiettivi[5].

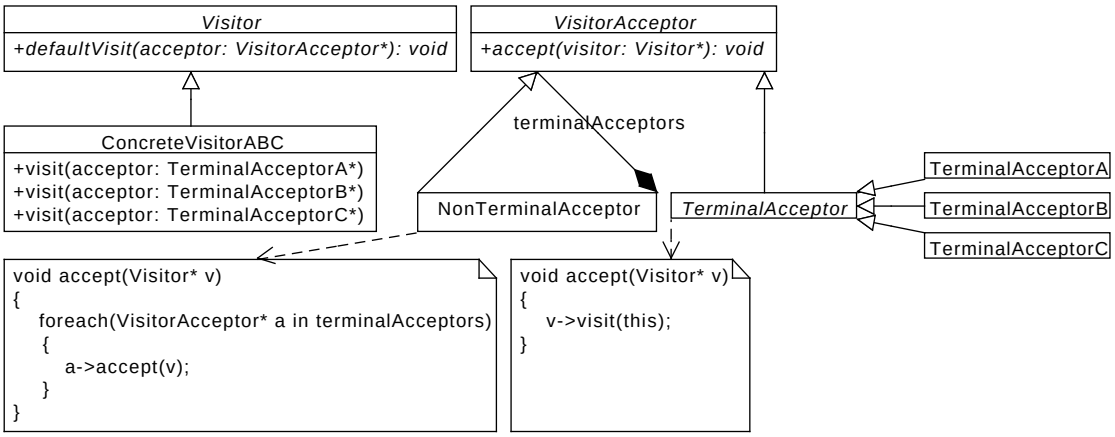


Figura A.4: Design pattern Visitor

[4] Il metodo `visit` invocato è proprio quello relativo al tipo dello specifico `VisitorAcceptor` invocante.

[5] Immaginiamo, ad esempio, un `MeanVisitor` ed un `MaxVisitor`: il primo ricava dalla visita della collezione la media degli elementi, mentre il secondo il massimo.

A.1.3 Queue

Il piccolo framework per la gestione delle code mostrato in figura A.5 è basato sul *design pattern Bridge* e consente di separare l' interfaccia di una coda dalla sua implementazione, permettendo di nasconderne i dettagli a tutti i client che non dispongono della conoscenza necessaria.

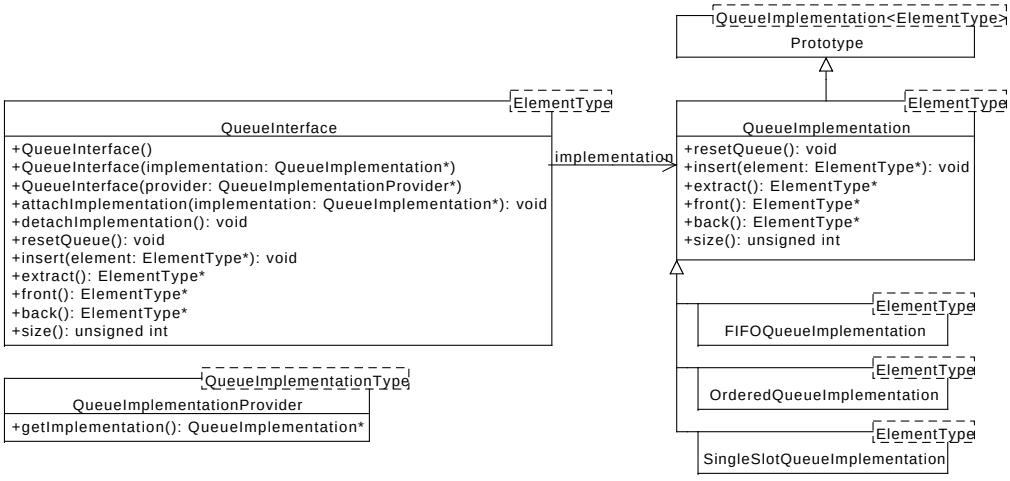


Figura A.5: Un semplice Queue framework basato su *Bridge design pattern*

A.1.4 EventManagement

In figura A.6 è riportato un semplice framework per *event handling*. È basato sul *design pattern Observer*: una `EventSource` è un `Subject` che gestisce una coda di `EventType`. Il tipo `EventType` deve derivare da o essere una specializzazione di `Event<SubjectIdentifierType, TimeType>`[6]. Quando la `EventSource` ritiene opportuno, può notificare ai suoi `Observer` che la propria coda di eventi è non vuota; gli `Observer` provvederanno, se necessario, a richiedere un clone di tale coda.

Il tipo `Event<SubjectIdentifierType, TimeType>` rappresenta un generico evento, ed è parametrizzabile rispetto al tipo del soggetto coinvolto nell' evento e al tipo di tempo (continuo, discreto...), mentre l' informazione relativa alla tipologia di evento risiede nel tipo stesso, nella particolare classe derivata da una specializzazione di `Event<SubjectIdentifierType, TimeType>`: ciò consente un utilizzo spinto del *polimorfismo* e della *metaprogrammazione template*.

[6] Un check a compile-time di tale asserzione viene effettuato attraverso la libreria *Boost Static Assert*, in congiunzione a *Boost Type Traits*.

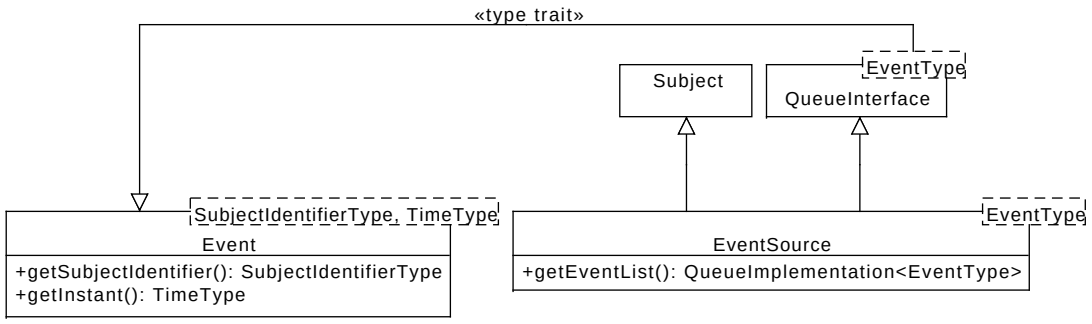


Figura A.6: Un semplice event management framework

A.2 Librerie esterne

A.2.1 GnuplotCpp

La libreria *GnuplotCpp* incapsula nel tipo `Gnuplot` le interazioni col processo `gnuplot`, utility che permette di plottare grafici. La comunicazione con `gnuplot`, in ambienti *UNIX-like*, avviene attraverso *UNIX pipe*.

A.2.2 Boost Program Options

La libreria *Boost Program Options* consente al programmatore di ottenere e gestire in maniera semplice e robusta le opzioni di esecuzione. La libreria fornisce le opzioni di esecuzione in maniera trasparente rispetto al metodo di inserimento (linea di comando, file di configurazione, ...), occupandosi al contempo di parsing, conversioni di tipo e controllo degli errori.

A.2.3 Boost Static Assert e Boost Type Traits

La libreria *Boost Static Assert* offre al programmatore la potentissima macro `BOOST_STATIC_ASSERT()` che consente di effettuare assertion check a compile-time. Questa macro può essere utilizzata, come è facile notare, solo su asserzioni decidibili a tempo di compilazione, quali le asserzioni su valori costanti o sui tipi.

La libreria *Boost Type Traits* consente al programmatore di creare in maniera semplice asserzioni sulle proprietà dei tipi, agevolando considerevolmente l' utilizzo di tecniche di *metaprogrammazione template*.

A.2.4 Boost Spirit

Boost Spirit è una potentissima libreria di parsing. Essa offre la possibilità di definire grammatiche *inline*, senza ricorrere a linguaggi diversi

dal C++, secondo il formalismo *Extended Backus Naur Form*. Tale risultato è ottenuto attraverso un uso massiccio di *polimorfismo*, *overloading*, *expression template* e *metaprogrammazione template*.

La generazione del parser avviene per composizione di parser più semplici, seguendo la grammatica, sino a giungere ai parser che riconoscono i simboli terminali. *Boost Spirit* è in grado di riconoscere un ampio sottoinsieme di *grammatiche libere dal contesto*, poiché genera dei parser $LL(\infty)$[7].

La libreria consente di collegare in maniera semplice la *sintassi* di un linguaggio alle *azioni semantiche* associate: ad ogni livello del *parse tree* generato dalla definizione della grammatica è possibile collegare, grazie alla libreria *Boost Phoenix*, una vasta gamma di *callable*: *funzioni*, *funtori*, *metodi*, *espressioni lambda*, *chiusure*.

[7]Un parser $LL(\infty)$ non ha *lookahead limitato*, ed è in grado di scegliere la giusta produzione riconoscendo l' appartenenza dei successivi token ad un *linguaggio regolare*.