



DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA,  
ELETTRONICA E SISTEMISTICA

Corso di Laurea Robotics and Automation Engineering

*Corso di Programmazione di Sistemi Tempo-Reali e Distribuiti*

## **Analisi e Casi di Studio: Reti di Petri, Timed Automata e Scheduling Real-Time**

Prof.  
Francesco Pupo

Giuseppe Coppola  
Mat. 263624

ANNO ACCADEMICO 25/26

# Indice

<b>Indice</b>	<b>1</b>
<b>1 Introduzione</b>	<b>3</b>
<b>2 Reti di Petri</b>	<b>4</b>
2.1 Introduzione alle Reti Di Petri . . . . .	4
2.1.1 Evoluzione della Rete . . . . .	5
2.1.2 Rappresentazione Matriciale . . . . .	6
2.1.3 Dinamica della Rete . . . . .	7
2.2 Caso di Studio: Catena di Montaggio . . . . .	8
2.2.1 Proprietà . . . . .	9
Raggiungibilità . . . . .	9
Binarietà . . . . .	10
Limitatezza . . . . .	11
Deadlock . . . . .	11
2.2.2 Rappresentazione Matriciale . . . . .	11
<b>3 Timed Automata</b>	<b>13</b>
3.1 Introduzione ai Timed Automata e Uppaal . . . . .	13
3.2 Caso di Studio: Gestione Energetica di un Robot Mobile . . . . .	14
3.2.1 Controllore . . . . .	15
3.2.2 Manager Missioni . . . . .	16
3.2.3 Stazione di Ricarica . . . . .	17
3.2.4 Simulazione e Analisi . . . . .	18
<b>4 Scheduling Real-Time</b>	<b>21</b>
4.1 Introduzione allo Scheduling Real-Time . . . . .	21
4.1.1 Strategie di Scheduling . . . . .	22
4.2 Caso di Studio: Centralina di Controllo di un Veicolo (ECU) . . . . .	22
4.2.1 Analisi di fattibilità . . . . .	23
4.2.2 Static Cycling Scheduling (SCS) . . . . .	23
Applicazione . . . . .	24
4.2.3 Rate Monotonic (RM) . . . . .	25
Analisi di Schedulabilità: Bound LL e Analisi Esatta . . . . .	26
Applicazione . . . . .	27
4.2.4 Earliest Deadline First (EDF) . . . . .	28
Applicazione . . . . .	28
4.2.5 Least Slack Time (LST) . . . . .	30

Applicazione . . . . .	30
<b>5 Conclusioni</b>	<b>32</b>
<b>Bibliografia</b>	<b>33</b>

# Capitolo 1

## Introduzione

Il presente elaborato si pone l’obiettivo di analizzare e approfondire le metodologie fondamentali per la progettazione, la modellazione e la verifica di **sistemi tempo-reali e distribuiti**.

Il lavoro è strutturato in tre parti principali, corrispondenti ai capitoli centrali del documento:

- **Reti di Petri.** Viene introdotto il formalismo grafico e matematico ideato da Carl Adam Petri, ideale per descrivere processi concorrenti e asincroni. Attraverso l’utilizzo del software *PIPE*, si analizza il caso di studio di una Catena di Montaggio industriale composta da macchinari di taglio, foratura e assemblaggio. L’analisi si concentra sulla verifica delle proprietà strutturali della rete, quali la raggiungibilità, la limitatezza e, in particolare, l’assenza di situazioni di blocco (Deadlock).
- **Timed Automata.** Si passa alla modellazione di sistemi tempo-dipendenti utilizzando il toolbox UPPAAL. Il caso di studio riguarda la Gestione Energetica di un Robot Mobile. Il sistema è modellato come una rete di automi paralleli che devono sincronizzarsi rigorosamente per alternare fasi di lavoro a fasi di ricarica, rispettando vincoli temporali stringenti per evitare il guasto del sistema.
- **Scheduling Real-Time.** Infine, si affronta il problema dell’allocazione della CPU ai processi in un sistema con vincoli temporali rigidi (Hard Real-Time). Viene preso in esame lo studio di una centralina di controllo di un veicolo (ECU), caratterizzato da tre processi principali con periodi e deadline differenti. Dopo un’analisi preliminare di fattibilità, vengono confrontati e applicati diversi algoritmi di scheduling: lo **Static Cyclic Scheduling** (SCS), il **Rate Monotonic** (RM), l’**Earliest Deadline First** (EDF) e il **Least Slack Time** (LST), verificandone la capacità di garantire il rispetto delle scadenze temporali.

Attraverso questi tre approcci, il progetto mira a fornire una visione completa del ciclo di vita della verifica di un sistema: dalla modellazione del flusso logico, alla gestione dei vincoli temporali, fino alla pianificazione operativa dell’esecuzione.

# Capitolo 2

## Reti di Petri

Una **Rete di Petri** è uno strumento grafico per la descrizione e l'analisi di processi concorrenti che hanno origine in sistemi a molti componenti (sistemi distribuiti). La rappresentazione grafica, insieme alle regole per la sua aggregazione e il suo raffinamento, fu inventata nell'agosto del 1939 dal tedesco *Carl Adam Petri*.

L'aspetto algebrico dei sistemi distribuiti fu descritto da Petri nel 1962 nella sua tesi di dottorato "*Communication with Automata*" (Petri, 1966), in cui si sosteneva che l'allora prevalente teoria degli automi dovesse essere sostituita da una nuova teoria che tenesse conto dei risultati della fisica moderna.

### 2.1 Introduzione alle Reti Di Petri

Formalmente, le Reti di Petri sono rappresentate da:

$$P/T = (P, T, F, W, M_0)$$

dove:

- **P** è un insieme finito di **posti** i quali possono essere visti come *contenitori di risorse*
- **T** è un insieme finito di **transizioni** le quali costituiscono gli eventi che possono accadere nel sistema
- **F** è la relazione di *flusso* o più comunemente rappresenta gli **archi** i quali sono orientati e collegano esclusivamente posti a transizioni (o viceversa)
- **W** è la funzione che assegna **pesi** non negativi agli archi.
- **M<sub>0</sub>** è la **marcatura iniziale**

Particolarmente, le Reti di Petri garantiscono le seguenti proprietà:

1. Un posto non può essere una transizione, e viceversa:

$$P \cap T = \emptyset$$

2. Una Rete di Petri deve essere composta da almeno un posto e almeno una transizione:

$$P \cup T \neq \emptyset$$

3. Gli archi della rete puntano sempre da posti a transizioni o viceversa:

$$F \subseteq (P \times T) \cup (T \times P)$$

4. La funzione peso associa un numero naturale positivo ad ogni arco:

$$W : F \rightarrow \mathbb{N}^*$$

5. La marcatura iniziale associa un numero di token ad ogni posto:

$$M_0 : P \rightarrow \mathbb{N}$$

I **token** sono fondamentali per l'abilitazione delle transizioni; infatti rappresentano le *risorse*.

Una transizione è *abilitata* se i suoi posti di ingresso contengono sufficienti risorse e, inoltre, in presenza di più transizioni abilitate, la scelta della transizione che scatta è **non deterministica**.

Lo scatto di una transizione è un evento *atomico istantaneo* che consuma token dai posti di ingresso e genera token nei posti di uscita.

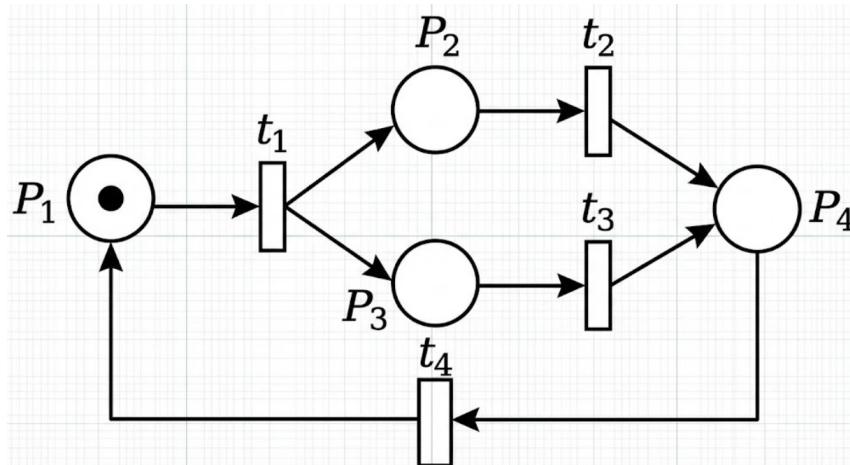


Figura 2.1: Esempio di Rete di Petri

### 2.1.1 Evoluzione della Rete

**Definizione 1.** *Preset* ( $Pre : X \rightarrow 2^X$ ) di un Posto/Transizione è dato da:

$$Pre(y) = \{z \in X \mid <z, y> \in F\}$$

**Definizione 2.** *Postset* ( $Post : X \rightarrow 2^X$ ) di un Posto/Transizione è dato da:

$$Post(y) = \{z \in X \mid <y, z> \in F\}$$

In questo caso  $X = P \cup T$  e  $2^X$  rappresenta l'insieme delle parti di  $X$  ovvero l'insieme i cui elementi sono tutti i possibili sottoinsiemi di  $X$  (compreso l'insieme vuoto e  $X$  stesso).

In definitiva, l'**abilitazione** di una transizione  $t$  in una marcatura  $M$ :

$$M[t] \iff (\forall p \in Pre(t)(M(p) \geq W(p, t)))$$

Nella Rete rappresentata nella Figura (2.1) l'unica transizione abilitata sarà  $t_1$ . Il Postset di  $t_1$  è  $\{P_1\}$ ; la marcatura di  $P_1$  è 1 e quindi risulta uguale al peso presente sull'arco che va dal posto  $P_1$  alla transizione  $t_1$ .

Avendo garantito l'abilitazione di  $t_1$ , la marcatura si modifica come segue  $M[t_1] > M'$ . Il tutto avviene mediante le seguenti regole:

1.  $\forall p \in Pre(t_1) - Post(t_1) \rightarrow M'(p) = M(p) - W(p, t_1)$
2.  $\forall p \in Post(t_1) - Pre(t_1) \rightarrow M'(p) = M(p) + W(t_1, p)$
3.  $\forall p \in Post(t_1) \cap Pre(t_1) \rightarrow M'(p) = M(p) - W(p, t_1) + W(t_1, p)$
4.  $\forall p \in P - (Pre(t_1) \cup Post(t_1)) \rightarrow M'(p) = M(p)$

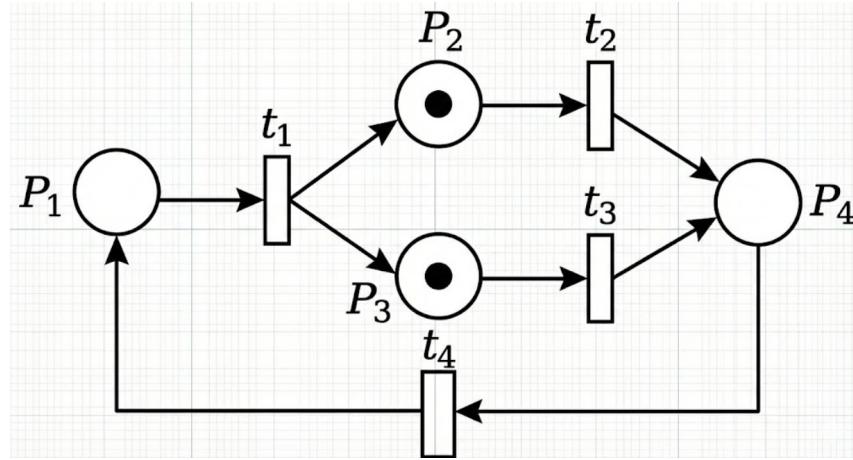


Figura 2.2: Rete di Petri dopo lo Sparo

### 2.1.2 Rappresentazione Matriciale

Una Rete di Petri può essere descritta in forma matriciale, in modo da risultare più compatta e utile per analizzarne la dinamica.

Si identificano i posti con gli indici da 1 a  $|P|$  (o da 0 a  $|P| - 1$ ) e le transizioni con gli indici da 1 a  $|T|$  (o da 0 a  $|T| - 1$ ).

La rappresentazione, quindi, è basata sull'utilizzo di due matrici di **incidenza**:

1. **Matrice di Input**  $I : |P| \times |T|$ , definita come:

$$I_{i,j} = W(p_i, t_j)$$

2. **Matrice di Output**  $O : |P| \times |T|$ , definita come:

$$O_{i,j} = W(t_j, p_i)$$

Inoltre, si definisce il **vettore marcatura**  $m : |P|$ .

L'effetto **netto** di scatto delle transizioni si può rappresentare con una terza matrice  $C : |P| \times |T|$  ed è definita come:

$$C = O - I$$

dove il generico elemento  $c_{i,j} \in \mathbb{Z}$ .

Infine, una transizione  $t_j$  è abilitata quando:

$$\forall i \mid p_i \in P, m[i] \geq I_{i,j}$$

Nel caso della Rete rappresentata nella Figura (2.1), una sua rappresentazione matriciale sarà:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad O = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad C = O - I = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix}$$

### 2.1.3 Dinamica della Rete

La dinamica della Rete si può rappresentare mediante un'equazione lineare:

$$m' = m + \sum_j C_{:,j} \cdot s_j$$

definita come **equazione fondamentale delle Reti P/T**.

In forma più compatta, l'equazione si può riscrivere come:

$$m' = m + C \cdot s$$

dove  $s$  rappresenta la **sequenza di scatto** delle transizioni.

Come si può verificare nella Figura (2.2), la sequenza di scatto sarà " $t_1$ " e, quindi, il vettore  $s$  è definito come:

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

La marcatura dopo lo scatto della sola transizione  $t_1$  è:

$$m' = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

## 2.2 Caso di Studio: Catena di Montaggio

Nella seguente sezione verrà analizzato il caso di una Catena di Montaggio all'interno di una fabbrica che lavora materie prime per poi assemblarle nel prodotto finale.



Figura 2.3: Catena di Montaggio

Il funzionamento è il seguente:

1. Il meccanismo ottiene in ingresso 5 materie prime da lavorare e modellare
2. Tramite un nastro trasportatore entra nella macchina per il **taglio**
3. Finito il taglio, tramite un altro nastro trasportatore, il materiale tagliato arriva nella macchina adibita alla **foratura**.
4. Infine, mediante un ulteriore nastro trasportatore, i 5 pezzi tagliati e forati, arrivano in un ultimo macchinario adibito all'**assemblaggio**.
5. Un ultimo nastro trasportatore porta in uscita il prodotto terminato.

La figura seguente descrive il funzionamento generale della Catena.

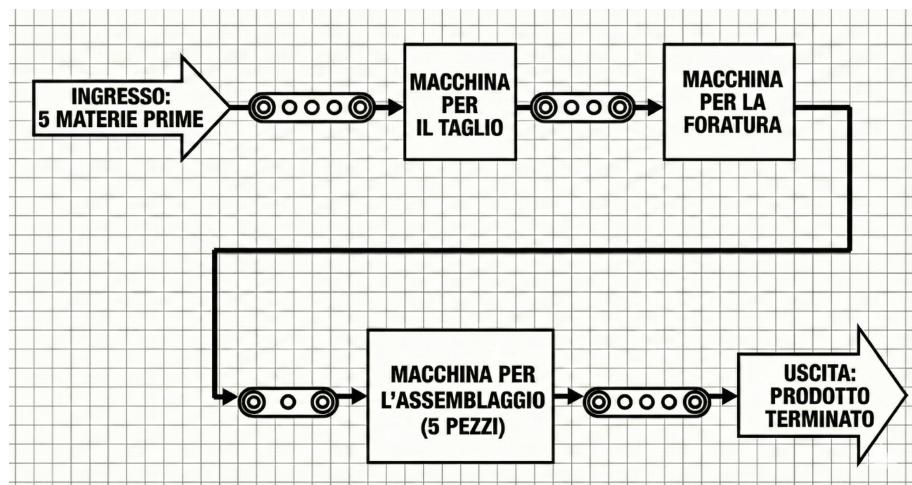


Figura 2.4: Grafico di Funzionamento

Per complicare ulteriormente la situazione, si assume che i macchinari sono soggetti a **guasti** che si possono verificare dopo un certo numero di utilizzi del macchinario o dopo un certo periodo completamente randomico.

Per la definizione della Rete di Petri si assume che:

- Non appena sono disponibili le 5 materie prime, si attiva subito il nastro trasportatore. In termini di Reti di Petri, la transizione analoga avrà priorità massima.
- I prodotti che arrivano al macchinario libero verranno processati subito; in termini di Reti di Petri, le transizioni analoghe presenteranno una priorità maggiore rispetto alle altre ma con priorità minore rispetto a quella descritta sopra.
- Infine, ogni prodotto che ha finito il suo processo di lavorazione passerà subito sul nastro trasportatore in uscita.

Utilizzando il software **PIPE**, si è ottenuta la seguente rete:

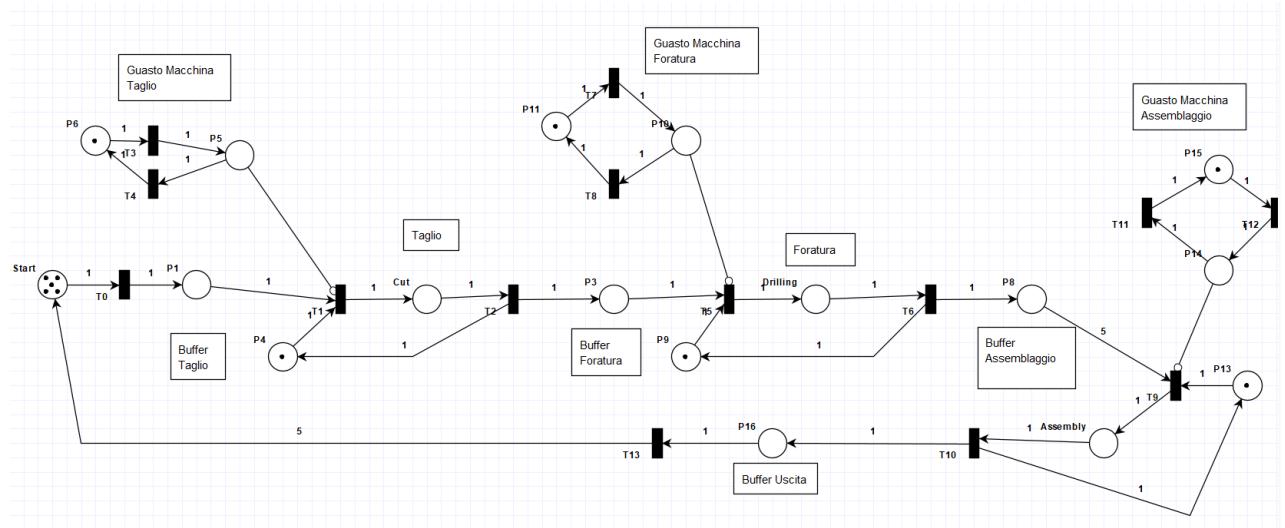


Figura 2.5: Petri Net (PIPE)

Da notare che ogni macchinario non può lavorare più elementi contemporaneamente eccetto l'ultimo adibito all'assemblaggio.

### 2.2.1 Proprietà

Si vogliono analizzare le proprietà che garantisce la rete in questione; soprattutto, utilizzando il software PIPE si verifica o meno la presenza di **Deadlock**.

#### Raggiungibilità

Una marcatura  $M'$  è raggiungibile da una marcatura  $M$  se esiste una sequenza di scatti in grado di trasformare la rete da  $M$  a  $M'$ .

La seguente rete risulta **raggiungibile**; si può dedurre ulteriormente dal suo *Grafo di Raggiungibilità* alla pagina successiva.

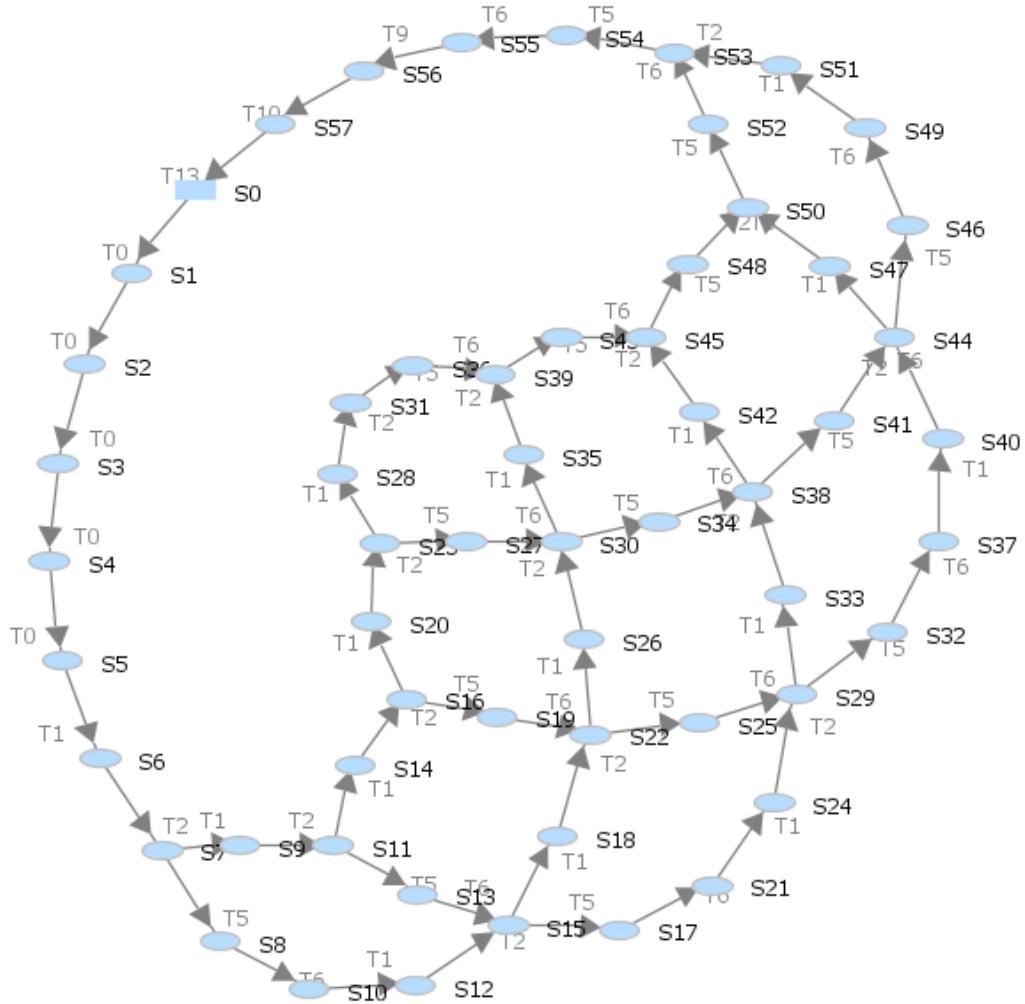


Figura 2.6: Grafo di Raggiungibilità (PIPE)

### Binarietà

Una Rete è binaria se in ogni suo posto, in ogni possibile marcatura, il numero di token è 0 o 1:

$$\forall M' \in R(P/T, M) \quad \forall p \in P(M'(p) \leq 1)$$

dove  $R(P/T, M)$  è l'*Insieme di Raggiungibilità* definito come tutte le marcature tali che:

$$\begin{cases} M \in R(P/T, M) \\ (M' \in R(P/T, M) \cap (\exists t \in T \ M'[t > M'']) \Rightarrow M'' \in R(P/T, M) \end{cases}$$

La rete in questione non sarà binaria in quanto più posti conterranno più di un token. Si può verificare facilmente grazie alla costruzione della rete in quanto inizia il suo funzionamento quando sono presenti in ingresso 5 materie prime, ovvero 5 token.

## Limitatezza

Una rete è limitata di ordine  $k$ , se in ogni marcatura raggiungibile il numero dei token in un qualsiasi posto non supera  $k$

$$\exists k \in \mathbb{N} \quad \forall M' \in R(P/T, M) \quad \forall p \in P, \quad M'(p) \leq k$$

La rete in questione risulta essere limitata in quanto il numero massimo di token non supera 5.

## Deadlock

### Petri net state space analysis results

<b>Bounded</b>	true
<b>Safe</b>	false
<b>Deadlock</b>	false

Figura 2.7: Verifica Proprietà (PIPE)

L'analisi in questione conferma l'assenza di *Deadlock*.

La Rete in esame non arriva in uno stato di bloccaggio poiché una volta che il meccanismo finisce tutto il suo processo, riattiva nuovamente il nastro trasportatore in ingresso facendo entrare in lavorazione le successive 5 materie prime.

### 2.2.2 Rappresentazione Matriciale

Come descritto in precedenza, una Rete di Petri può essere rappresentata mediante 3 matrici di incidenza.

- **Matrice di Input ( $I$ )**

Backwards incidence matrix $I$														
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
<b>Start</b>	1	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>P1</b>	0	1	0	0	0	0	0	0	0	0	0	0	0	0
<b>Cut</b>	0	0	1	0	0	0	0	0	0	0	0	0	0	0
<b>P3</b>	0	0	0	0	0	1	0	0	0	0	0	0	0	0
<b>P4</b>	0	1	0	0	0	0	0	0	0	0	0	0	0	0
<b>P5</b>	0	0	0	0	1	0	0	0	0	0	0	0	0	0
<b>P6</b>	0	0	0	1	0	0	0	0	0	0	0	0	0	0
<b>Drilling</b>	0	0	0	0	0	0	1	0	0	0	0	0	0	0
<b>P8</b>	0	0	0	0	0	0	0	0	0	5	0	0	0	0
<b>P9</b>	0	0	0	0	0	1	0	0	0	0	0	0	0	0
<b>P10</b>	0	0	0	0	0	0	0	0	1	0	0	0	0	0
<b>P11</b>	0	0	0	0	0	0	0	1	0	0	0	0	0	0
<b>Assembly</b>	0	0	0	0	0	0	0	0	0	0	1	0	0	0
<b>P13</b>	0	0	0	0	0	0	0	0	0	1	0	0	0	0
<b>P14</b>	0	0	0	0	0	0	0	0	0	0	1	0	0	0
<b>P15</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0
<b>P16</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figura 2.8: Matrice Input (PIPE)

- Matrice di Output ( $O$ )

	Forwards incidence matrix $I^+$													
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
Start	0	0	0	0	0	0	0	0	0	0	0	0	0	5
P1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Cut	0	1	0	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	1	0	0	0	0	0	0	0	0	0	0	0
P4	0	0	1	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	1	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Drilling	0	0	0	0	0	1	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	1	0	0	0	0	0	0	0
P9	0	0	0	0	0	0	0	1	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	1	0	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Assembly	0	0	0	0	0	0	0	0	0	0	1	0	0	0
P13	0	0	0	0	0	0	0	0	0	0	1	0	0	0
P14	0	0	0	0	0	0	0	0	0	0	0	0	1	0
P15	0	0	0	0	0	0	0	0	0	0	0	1	0	0
P16	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Figura 2.9: Matrice Output (PIPE)

- Matrice Combinata ( $C$ )

	Combined incidence matrix $I$													
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
Start	-1	0	0	0	0	0	0	0	0	0	0	0	0	5
P1	1	-1	0	0	0	0	0	0	0	0	0	0	0	0
Cut	0	1	-1	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	1	0	0	-1	0	0	0	0	0	0	0	0
P4	0	-1	1	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	1	-1	0	0	0	0	0	0	0	0	0
P6	0	0	0	-1	1	0	0	0	0	0	0	0	0	0
Drilling	0	0	0	0	0	1	-1	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	1	0	0	-5	0	0	0	0
P9	0	0	0	0	0	-1	1	0	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	1	-1	0	0	0	0	0
P11	0	0	0	0	0	0	0	-1	1	0	0	0	0	0
Assembly	0	0	0	0	0	0	0	0	0	1	-1	0	0	0
P13	0	0	0	0	0	0	0	0	0	-1	1	0	0	0
P14	0	0	0	0	0	0	0	0	0	0	-1	1	0	0
P15	0	0	0	0	0	0	0	0	0	0	1	-1	0	0
P16	0	0	0	0	0	0	0	0	0	0	1	0	0	-1

Figura 2.10: Matrice  $C$  (PIPE)

# Capitolo 3

## Timed Automata

**UPPAAL** è un toolbox per la verifica di sistemi real-time (in tempo reale), sviluppato congiuntamente dall’Università di Uppsala e dall’Università di Aalborg.

I **Timed Automata** costituiscono un formalismo per la modellazione e la verifica di sistemi tempo dipendenti e, inoltre, sono un’estensione dei classici automi a stati finiti.

### 3.1 Introduzione ai Timed Automata e Uppaal

Un **Timed Automata** è una macchina a stati finiti estesa con *variabili clock* i quali progrediscono in modo sincrono.

In **UPPAAL**, un sistema è modellato come una rete di diversi automi temporizzati in parallelo. Il modello è ulteriormente esteso con variabili discrete limitate che fanno parte dello stato. Queste variabili sono usate come nei linguaggi di programmazione: vengono lette, scritte e sono soggette a comuni operazioni aritmetiche.

**Definizione 3.** *Un Timed Automata è una tupla  $(L, l_0, C, A, E, I)$  dove:*

- $L$  è l’insieme delle **locazioni**
- $l_0$  è la **locazione iniziale**
- $C$  è l’insieme dei **clock**
- $A$  è l’insieme delle **azioni, co-azioni e le  $\tau$ -azione interne**
- $E \subseteq L \times A \times B(C) \times 2^C \times L$  è l’insieme di **archi tra locazioni e un’azione**
- $I : L \rightarrow B(C)$  **assegna invarianti a locazioni**

Si ha che, se  $C$  è l’insieme di clock,  $B(C)$  denota le **congiunzioni** che si possono formare utilizzando confronti di clock con numeri naturali.

Un Timed Automata evolve in un insieme di possibili stati  $S$  a seguito di transizioni che possono essere **delay** o **action**.

Inoltre, in Uppaal, un modello può essere ispezionato (testato) in simulazione oppure le sue proprietà possono essere verificate usando *model checker*; le proprietà che si possono analizzare sono **raggiungibilità, safety e liveness**.

## 3.2 Caso di Studio: Gestione Energetica di un Robot Mobile

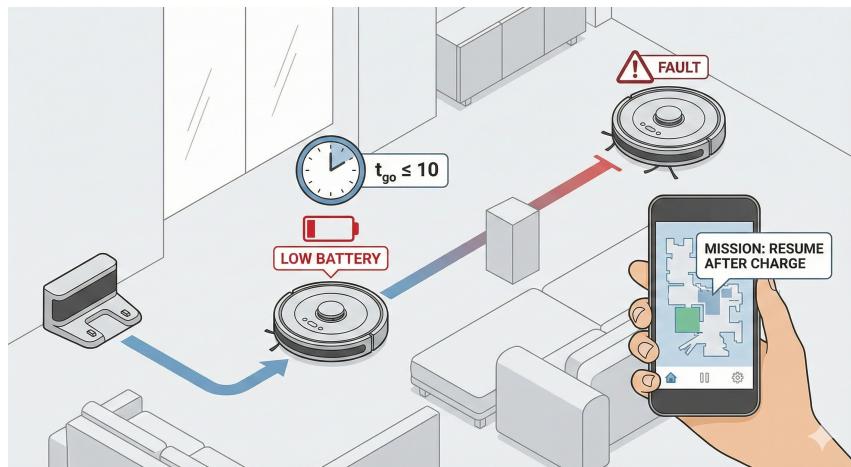


Figura 3.1: Ambiente di Lavoro di un Robot Aspirapolvere

Il sistema in questione rappresenta il comportamento di un robot mobile incaricato di eseguire missioni.

Basandosi su un sistema a condizioni reali, considerando un robot aspirapolvere, le missioni in questione sono i vari punti da raggiungere nell’ambiente circostante affinché avvenga una pulizia corretta.

L’obiettivo del modello è quello di verificare la correttezza delle missioni, la gestione delle risorse energetiche e la robustezza del sistema in caso di ritardi.

Il funzionamento del sistema consiste nelle seguenti fasi:

1. All’inizio il robot è sulla base di ricarica (dock), carico e in attesa della programmazione o che l’utente prema il bottone *”Start”*.
2. Una volta stabilita la *”missione”* corrente, ovvero il punto che deve si raggiungere, il robot si muove e contemporaneamente aspira raggiungendo il punto stabilito ovviamente consumando batteria.
3. Le missioni vengono stabilite e compiute ciclicamente finché la batteria del robot stesso raggiunge una percentuale di ricarica di circa 15%. Per una questione di robustezza del sistema, il robot, a questa percentuale, si avvia verso la base di ricarica così da caricare la sua batteria.
4. Una volta caricata la batteria, il robot non resta fermo, ma torna esattamente dove aveva interrotto per finire la stanza.

La modellazione in **Timed Automata** consiste nella definizione di 3 automi differenti i quali interagiscono parallelamente:

- **Robot\_Controller:** attore principale che esegue il lavoro e monitora il proprio stato energetico.

- **Mission\_Manager**: controllore logico il quale assegna i compiti e le missioni e monitora il loro stato.
- **Charging\_Station**: risorsa condivisa necessaria per il ripristino dell'energia del robot.

La comunicazione tra il tutto avviene mediante i canali definiti nella figura successiva:

```
// Place global declarations here.
chan start_mission, wait_mission, end_mission;
chan request_charge, start_charge, end_charge;
chan go_work;
```

Figura 3.2: Dichiarazioni Globali (UPPAAL)

Così da avere visibilità globale, i canali vengono definiti nella sezione dedicata alle dichiarazioni globali.

### 3.2.1 Controllore

È l'automa più complesso, dotato di 4 clock locali:

- **Durata del lavoro (missione)**:  $t\_work$
- **Consumo cumulativo dell'energia**:  $t\_battery$
- **Durata della ricarica**:  $t\_charge$
- **Timeout spostamenti**:  $t\_go$

In *UPPAAL* si definiscono le seguenti variabili nella sezione dedicata alle dichiarazioni locali del template adibito al controllore:

```
// Place local declarations here.
clock t_work, t_battery, t_charge, t_go;
```

Figura 3.3: Dichiarazioni Locali (UPPAAL)

Il robot appena "acceso" passa dallo stato **Wait** allo **Idle**, inizializzando il primo ciclo di batteria.

Principalmente, il comportamento del robot si può suddividere in due macro-cicli:

- **Ciclo di lavoro**: partendo dallo stato **Idle**, alla ricezione del segnale *start\_mission* transita nello stato **Working** nel quale svolge il computo fintanto che la batteria lo permette.

Se il lavoro termina (dopo 10 unità di tempo,  $t\_work \geq 10$ ) e la batteria è sufficiente, invia un segnale *end\_mission* e torna in **Idle**.

- **Ciclo di ricarica** (Gestione Energetica): durante il ciclo di lavoro, se il clock **t\_battery** raggiunge la soglia critica di 100 unità, il robot interrompe forzatamente la missione.

- Segnalazione:** invia *wait\_mission* al *Manager* e transita nello stato **Low\_Battery**
- Richiesta:** tenta di accedere alla stazione di ricarica inviando *request\_charge*. Se la stazione è libera, passa a **Going\_to\_Charge**.
- Timeout/Guasti:** se il robot non riesce a richiedere la ricarica o a connettersi entro 10 unità di tempo ( $t\_go > 10$ ), scatta un timeout che porta il sistema nello stato irreversibile di **Fault**. Questo potrebbe indicare una situazione di guasto per batteria esaurita.
- Ricarica:** se la connessione avviene (*start\_charge*), entra in **Charging**; la ricarica dura tra le 25 e le 30 unità di tempo.
- Ripresa:** al termine invia un segnale *end\_charge*, si azzera l'utilizzo della batteria ( $t\_battery := 0$ ), notifica il *Manager* di essere pronto tramite *go\_work* e riprende il lavoro precedente ritornando nello stato **Working**.

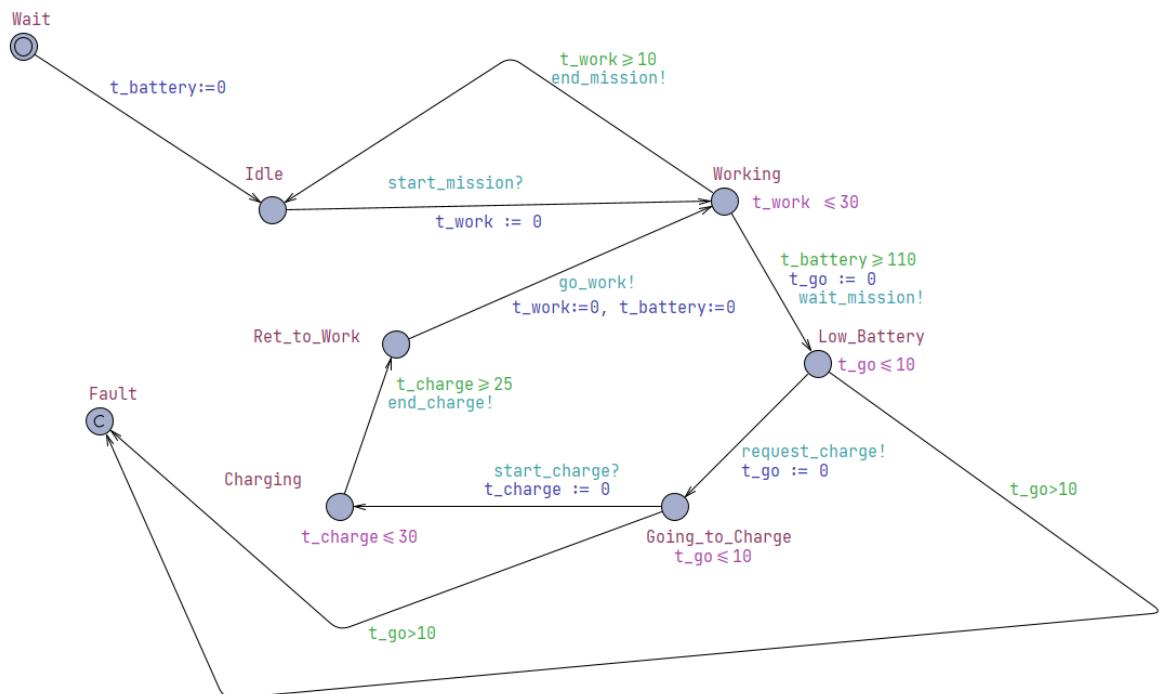


Figura 3.4: Template Robot\_Controller (UPPAAL)

### 3.2.2 Manager Missioni

Questo Timed Automata gestisce e supervisiona il ciclo di vita delle missioni.

Come dichiarazione locale si definisce il singolo clock:

```
// Place local declarations here.
clock t_mission;
```

Figura 3.5: Dichiarazioni Locali (UPPAAL)

Lo stato iniziale è **Wait**; dopo un breve setup (**Assign\_Mission**), invia un segnale al robot *start\_mission*.

Nello stato **Mission\_Assigned**, il manager attende la conclusione del lavoro da parte del robot.

Infine, è presente una sezione adibita alle gestione delle interruzioni. Se il robot segnala batteria scarica mediante *wait\_mission*, il Manager transita in uno stato di attesa **Charging** sospendendo momentaneamente la missione.

Una volta che il robot finisce il suo ciclo di ricarica viene segnalato mediante *go\_work* la ripresa della missione.

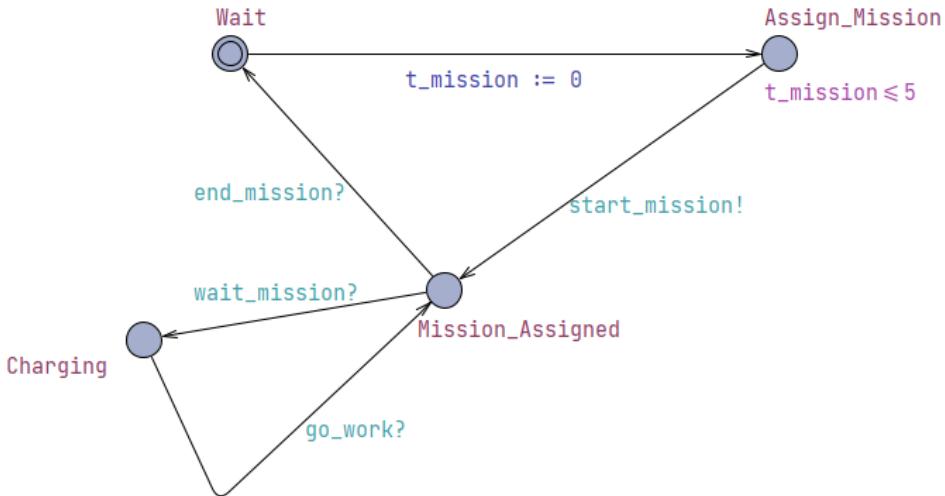


Figura 3.6: Template Mission\_Manager (UPPAAL)

### 3.2.3 Stazione di Ricarica

Questo Timed Automata modella la risorsa di ricarica con un meccanismo di esclusione (occupato/libero).

Anche in questo caso, solamente il clock viene definito nella sezione di dichiarazione locale:

```
// Place local declarations here.
clock t_station;
```

Figura 3.7: Dichiarazioni Locali (UPPAAL)

Si parte dallo stato **Free**; alla ricezione del segnale *request\_charge*, passa in **Wait\_Request** simulando un tempo tecnico di preparazione di 4 unità di tempo.

Per confermare l'inizio della ricarica manda un segnale *start\_charge* e passa nello stato **Occupied**.

Infine, la stazione rimane occupata finché il robot non segnala la fine tramite *end\_charge*. Alla ricezione di fine ricarica, la stazione ritorna ad essere libera e quindi nello stato **Free**.

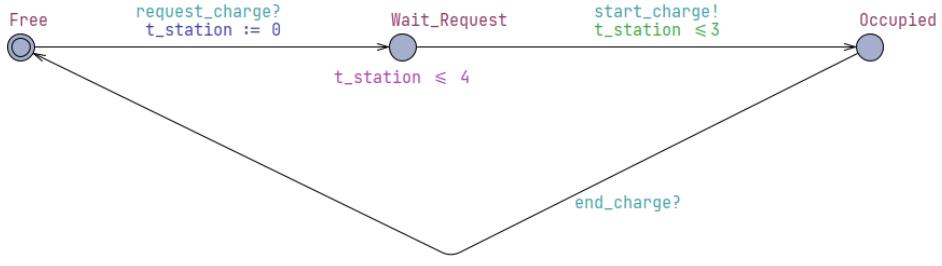


Figura 3.8: Template Charging\_Station (UPPAAL)

### 3.2.4 Simulazione e Analisi

Il corretto funzionamento del sistema è garantito da vincoli temporali ovvero **Invariante** e **Guardie** sulle transizioni:

- **Urgenza della Ricarica:** gli stati **Low\_Battery** e **Going\_to\_Charge** possiedono un invariante stretto  $t\_go \leq 10$ . Questo modella fisicamente l'autonomia residua; se il robot non raggiunge la stazione di carica entro questo tempo il sistema fallisce.
- **Durata del Lavoro:** lo stato **Working** ha un invariante  $t\_work \leq 30$ , forzando il robot a concludere o, eventualmente, interrompere la missione entro un tempo limitato.
- **Sicurezza della Ricarica:** la stazione di ricarica non può iniziare la ricarica istantaneamente ma richiede un tempo di setup gestito dal clock  $t\_station$  nello stato **Wait\_Request**.

A questo punto è possibile procedere con una simulazione randomica tramite il software. La simulazione è prevista per un certo periodo di tempo come mostrato in Figura (3.9). In sintesi, si può verificare che la traccia di simulazione prodotta certifica la correttezza del protocollo di comunicazione. Non si verificano deadlock né livelock (cicli infiniti improduttivi).

Il sistema alterna correttamente fasi di lavoro ininterrotto a fasi di manutenzione energetica, rispettando i vincoli temporali imposti e garantendo la coerenza degli stati tra i tre automi paralleli.

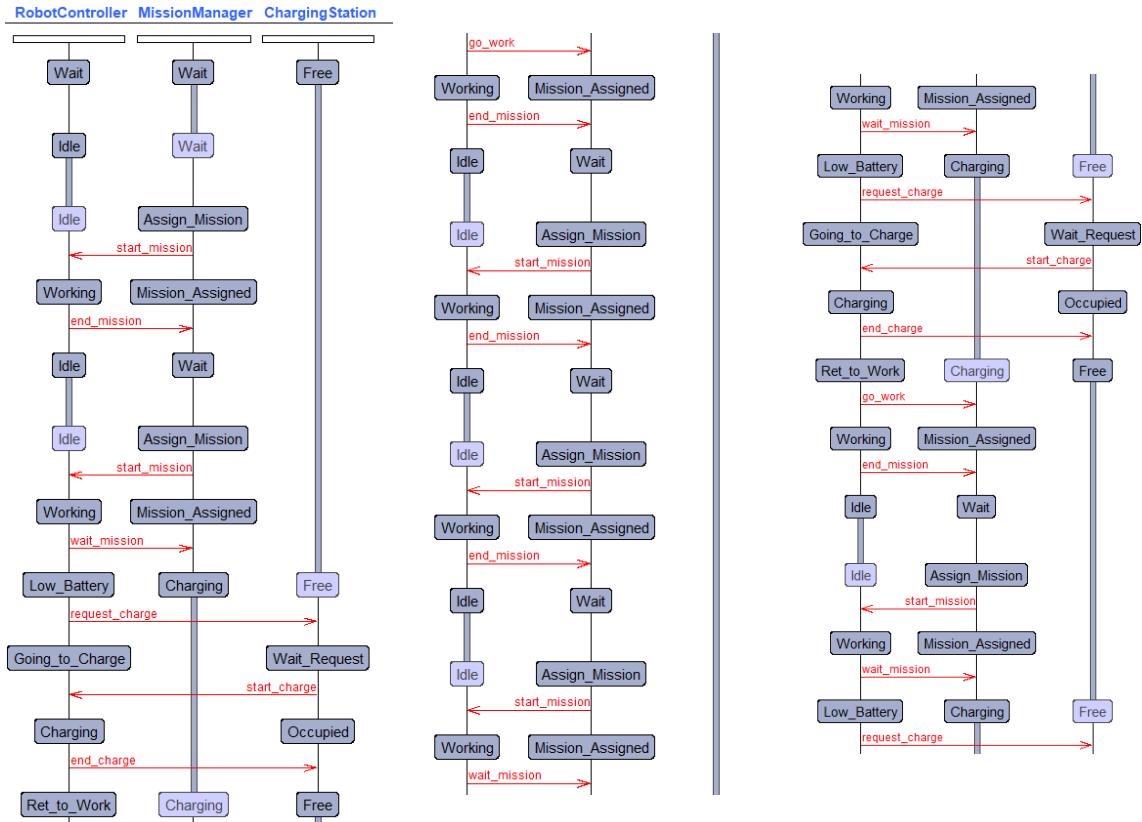


Figura 3.9: Simulazione del Sistema (UPPAAL)

In fase finale, per testare la **robustezza** e le **performance** del sistema, si possono costruire delle *query*.

Il tutto viene mostrato in Figura (3.10).

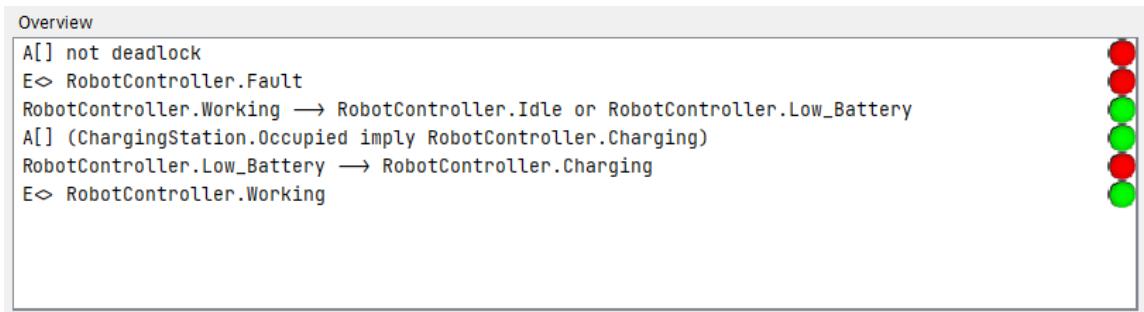


Figura 3.10: Query (UPPAAL)

Si può verificare:

1. **Assenza di deadlock:** nel modello è presente lo stato Fault. Questo stato rappresenta una condizione terminale che si verifica quando il robot esaurisce completamente l'energia prima di raggiungere la stazione di ricarica. Poiché dallo stato Fault non esistono transizioni uscenti il sistema entra in uno stato di arresto permanente.
2. **Raggiungibilità del Fault (Safety):** questa query verifica se esiste un percorso per cui il robot fallisce. Avendo risultato negativo, il robot non fallisce mai e quindi il sistema è affidabile al 100%.

3. **Ciclo di vita della Missione:** in questa query si verifica se quando il robot lavora, prima o poi si scarichi. La risposta sarà positiva e quindi questa situazione si può verificare durante il ciclo di lavoro.
4. **Utilizzo corretto della Stazione:** Verifica se la logica di sincronizzazione non sia disallineata, ad esempio se la stazione pensa di caricare, ma il robot è altrove. Quindi, avendo risposta positiva, si ha che la stazione occupata implica che il robot si trova sicuramente nello stato di ricarica.
5. **Liveness (Response):** si verifica se si garantisce che ogni volta che il robot si trova in stato di batteria scarica riesca sempre a raggiungere lo stato di ricarica. Poiché esiste la possibilità che il robot aspetti troppo tempo o non riesca a comunicare con la stazione, scatta il timeout e va in Fault. Una volta in Fault, il robot si ferma e non arriverà mai in Charging.
6. **Raggiungibilità del "lavoro":** serve a dimostrare che il sistema "parte" e fa quello che deve fare. Avendo un riscontro positivo da parte della query si può confermare che la catena di attivazione iniziale è modellata correttamente.

# Capitolo 4

## Scheduling Real-Time

Lo **Scheduling in Tempo Reale** rappresenta una componente fondamentale nell'ingegneria del software per sistemi critici. A differenza dei sistemi operativi general-purpose, dove l'obiettivo primario è spesso la massimizzazione del throughput o la minimizzazione del tempo medio di risposta, nei sistemi Real-Time la correttezza del sistema dipende non solo dal risultato logico della computazione, ma anche dal momento in cui questo risultato viene prodotto.

L'obiettivo dello Scheduling Real-Time è garantire la predicitività temporale, assicurando che i processi (o task) completino la loro esecuzione entro vincoli temporali prefissati, noti come deadline.

### 4.1 Introduzione allo Scheduling Real-Time

Il componente fondamentale dello Scheduling è il **processo**.

Per analizzare e gestire le tempistiche, i processi vengono modellati attraverso parametri specifici.

Ogni processo è definito come:

$$P = (C, T, D)$$

dove

- **Computation Time ( $C$ )**: tempo di esecuzione nel caso peggiore
- **Period ( $T$ )**: periodo di attivazione o la distanza minima tra due attivazioni successive.
- **Relative Deadline ( $D$ )**: intervallo di tempo entro cui il task deve terminare la sua esecuzione rispetto al momento della sua attivazione.

Inoltre, i processi vengono classificati in base alla regolarità delle loro attivazioni:

1. **Periodici**: si attivano a intervalli regolari di tempo  $T$ .
2. **Sporadici**: sono guidati da eventi con una distanza minima garantita tra due occorrenze consecutive.
3. **Aperiodici**: hanno attivazioni imprevedibili e possono verificarsi in successione ravvicinata.

Lo scheduling dei processi è organizzato dividendo l'asse temporale in intervalli della stessa lunghezza o **frame** (corrispondenti al tempo di clock del processore). La lunghezza minima di un intervallo è detta **minor cycle time** (mct).

Inoltre, lo scheduling si ripete dopo un tempo detto **major cycle time** (MCT). Ciò vuol dire che, se non ci sono errori, al termine di ogni MCT ogni processo deve essere stato eseguito completamente almeno una volta:

$$MCT = mcm(T_i) \quad i = 1, \dots, n \quad (4.1)$$

con  $n$  il numero totale dei processi.

#### 4.1.1 Strategie di Scheduling

Esistono diversi approcci algoritmici per allocare la CPU ai vari processi in modo da soddisfare i vincoli temporali. Le due macro-categorie principali sono:

- **Static Cyclic Scheduling (SCS):** Una tecnica basata su una tabella di scheduling definita a priori (off-line). L'asse temporale è diviso in intervalli fissi (frame) e la sequenza di esecuzione è rigida. Questo approccio offre altissima predicitività ma scarsa flessibilità ai cambiamenti.
- **Scheduling Dinamico a Priorità:** I processi sono scelti in base a priorità assegnate, che possono essere:
  1. **Statiche:** dove, ad esempio, la priorità è assegnata in base alla frequenza di attivazione
  2. **Dinamiche:** dove la priorità può cambiare nel tempo.

## 4.2 Caso di Studio: Centralina di Controllo di un Veicolo (ECU)

Si vuole analizzare la gestione dei processi principali del sistema di sicurezza e controllo di un'automobile moderna.

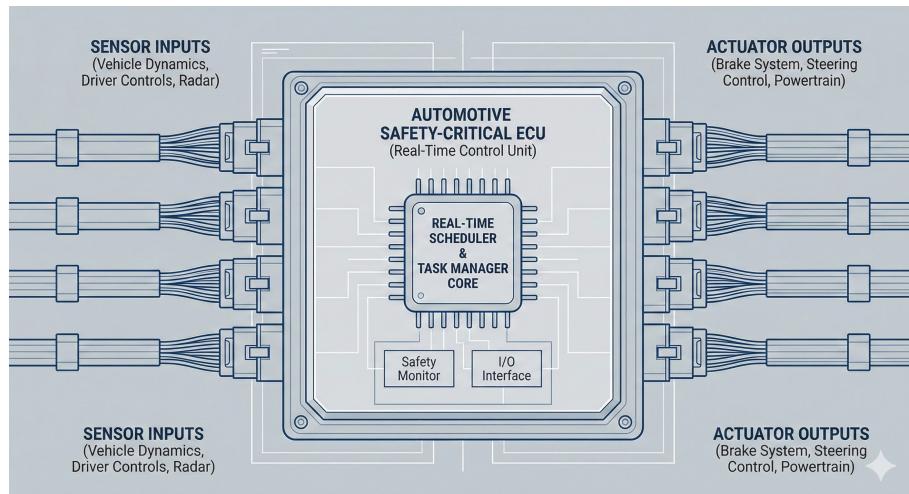


Figura 4.1: Sistema di controllo (ECU)

Si considerano i seguenti processi:

ID	Ruolo	C [ms]	T [ms]	D [ms]	Descrizione
$P_1$	ABS/Controllo Stabilità (ESP)	2	6	6	Lettura sensori ruote e frenata.
$P_2$	Quadro Strumenti (Display)	2	12	12	Aggiornamento tachimetro e giri motore.
$P_3$	Ricalcolo rotta GPS/ Telemetria	8	24	24	Calcolo del percorso o invio dati alla casa madre.

Tabella 4.1: Processi del modello

Si può verificare come i processi sono indipendenti fra loro e la *relative deadline* corrisponde con il *computation time*.

In particolare, il processo  $P_1$  deve avvenire rapidissimamente e molto spesso. Se l'auto ad esempio slitta, il sistema deve reagire in millisecondi.

Poiché il guidatore deve vedere la velocità aggiornata fluidamente, il processo  $P_2$  è il responsabile. Se però salta un aggiornamento per qualche millisecondo non è un problema. Infine, il processo  $P_3$  è adibito ad operazioni pesanti le quali richiedono calcoli complessi. Infatti, è il processo meno prioritario ma la cosa più importante è che i risultati arrivino in un tempo ragionevole (24 ms).

#### 4.2.1 Analisi di fattibilità

Come passo preliminare, si deve determinare il **Fattore di Utilizzazione** della CPU ( $U$ ). A livello pratico, rappresenta la frazione di tempo che il processore deve dedicare all'esecuzione dei task. In termini matematici:

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

Calcolando con i processi del sistema preso in esame:

$$U = \frac{2}{6} + \frac{2}{12} + \frac{8}{24} = \frac{20}{24} = 0.83 < 1$$

Avendo un fattore di utilizzazione **minore di 1**, il carico di lavoro è teoricamente gestibile dal processore.

Si potrà verificare nelle sezioni successive che  $U$  rappresenterà una condizione necessaria o necessaria e sufficiente.

Quindi, in definitiva, la condizione necessaria affinché uno schedule sia fattibile (indipendentemente dall'algoritmo scelto) risulta verificata.

#### 4.2.2 Static Cycling Scheduling (SCS)

Come definito in precedenza l'asse temporale si divide in **MCT** la quale rappresenta la durata della tabella di scheduling e si definisce come in (4.1); la particolarità è che ogni processo viene eseguito almeno una volta.

L'altra divisione è **mct** ovvero la suddivisione in intervalli più piccoli a lunghezza fissa.

La parte fondamentale del seguente algoritmo è la scelta della lunghezza del **frame** (mct). Per la sua definizione, si devono soddisfare i seguenti tre vincoli matematici:

1. **Divisibilità:**  $MCT$  deve essere divisibile per  $mct$
2. **Vincolo di Deadline:** il frame deve essere abbastanza breve da garantire che nessun processo manchi la sua deadline:

$$mct \leq D_i \quad i = 1, \dots, n$$

3. **Vincolo di Frame:** si deve garantire che un processo completi la sua esecuzione entro la sua deadline anche nel caso peggiore di sfasamento tra il suo arrivo e l'inizio del frame:

$$mct + (mct - MCD(mct, T_i)) \leq D_i \quad i = 1, \dots, n \quad (4.2)$$

## Applicazione

Come primo passo si calcola l'**MCT**:

$$MCT = mcm(T_1, T_2, T_3, T_4) = mcm(6, 12, 24) = 24$$

Per quanto riguarda la scelta dell'**mct**, si ha che sicuramente dovrà essere maggiore o uguale a 2 a causa del processo  $P_1$  e  $P_2$  e ai loro *Computation Time*.

Si vuole ora garantire il vincolo di divisibilità. Sapendo che i divisori di 24 sono  $\{1, 2, 3, 4, 6, 8, 12, 24\}$  e avendo  $mct \geq 2$ , i possibili candidati saranno:

$$\{2, 3, 4, 6, 8, 12, 24\}$$

L'ulteriore vincolo da rispettare è quello sulla deadline. A causa del processo  $P_1$  si avrà:

$$mct \leq 6$$

In definitiva, i possibili candidati risultano essere i seguenti:

$$\{2, 3, 4, 6\}$$

Infine, si vuole garantire il vincolo (4.2).

- Test con  $mct = 6$ :

$$\begin{aligned} P1 : \quad 6 + (6 - MCD(6, 6)) &\leq 6 \rightarrow 6 \leq 6 & (OK) \\ P2 : \quad 6 + (6 - MCD(6, 12)) &\leq 6 \rightarrow 6 \leq 12 & (OK) \\ P3 : \quad 6 + (6 - MCD(6, 24)) &\leq 6 \rightarrow 6 \leq 24 & (OK) \end{aligned}$$

Questo valore sarà valido.

- Test con  $mct = 4$ :

$$\begin{aligned} P1 : \quad 4 + (4 - MCD(4, 6)) &\leq 6 \rightarrow 4 + (4 - 2) \leq 6 \rightarrow 6 \leq 6 & (OK) \\ P2 : \quad 4 + (4 - MCD(4, 12)) &\leq 12 \rightarrow 4 \leq 4 & (OK) \\ P3 : \quad 4 + (4 - MCD(4, 24)) &\leq 24 \rightarrow 4 \leq 24 & (OK) \end{aligned}$$

Questo valore sarà valido.

- Test con  $mct = 3$ . Se verrà garantito il vincolo per  $P_1$  allora verrà garantito per gli altri processi avendo deadline più ampie.

$$P_1 : 3 + (3 - MCD(3, 6)) \leq 6 \rightarrow 3 \leq 6 \quad (OK)$$

Questo valore sarà valido.

- Test con  $mct = 2$ . Questo valore sarà sicuramente valido ma comporta ad avere troppi tick e quindi un massimo overhead di context switch.

In conclusione, si sceglie il valore di  $mct$  più alto possibile così da minimizzare i context switch. Pertanto, la scelta ottimale sarà:

$$MCT = 24$$

$$mct = 6$$

Di seguito un applicazione MATLAB che simula lo scheduling:

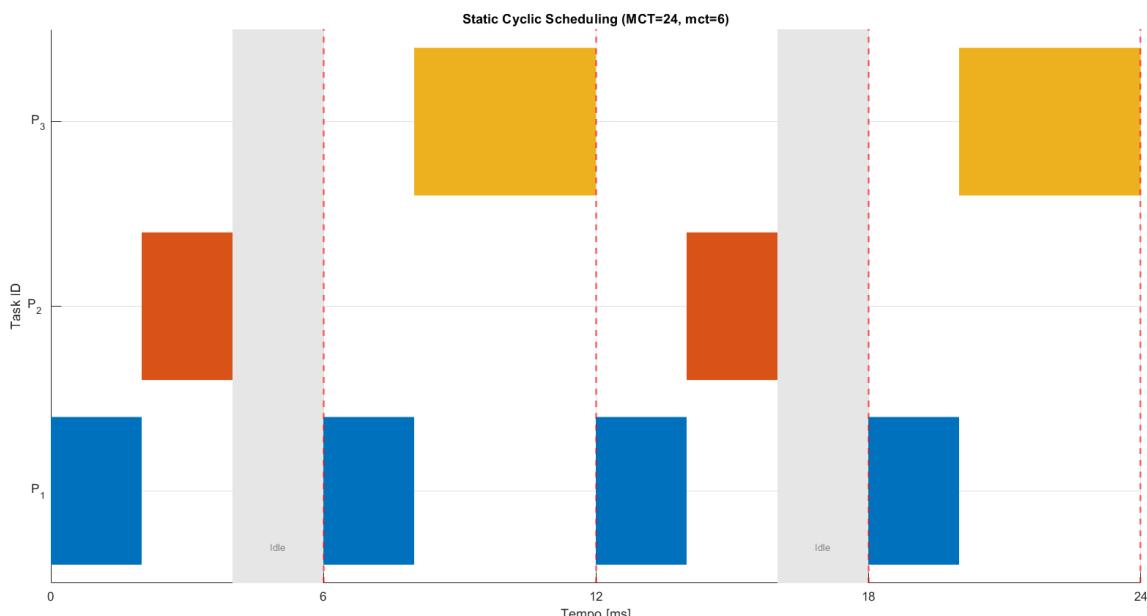


Figura 4.2: Static Cycling Scheduling (MATLAB)

### 4.2.3 Rate Monotonic (RM)

**Rate Monotonic (RM)** è l'algoritmo di riferimento per lo scheduling a **priorità statiche** ( $\pi$ ).

Le priorità vengono assegnate staticamente (off-line) prima dell'esecuzione e non cambiano durante la vita del sistema.

Rate Monotonic è un algoritmo **pre-emptive**, ovvero che ad ogni istante, il processore viene assegnato al task pronto con priorità più alta e, se arriva un task a priorità maggiore mentre ne sta eseguendo uno a priorità minore, il task corrente viene immediatamente interrotto per lasciare spazio a quello più prioritario.

## Analisi di Schedulabilità: Bound LL e Analisi Esatta

La condizione necessaria affinché uno schedule sia fattibile è già stata dimostrata. Nel caso del RM, si vuole verificare anche la condizione sufficiente; grazie a **Liu & Layland** si può verificare la condizione mediante la seguente formula matematica:

$$U \leq n \cdot \left( 2^{\frac{1}{n}} - 1 \right)$$

Questo rappresenta il **bound LL** dove  $n$  è il numero di processi del sistema.

Nel modello in questione:

$$0.83 \leq 3 \cdot \left( 2^{\frac{1}{3}} - 1 \right) = 0.77$$

La condizione sufficiente non è dimostrata e quindi non si può garantire la schedulabilità con questo metodo.

È necessario quindi procedere con l'analisi esatta dei **tempi di risposta** sviluppata da **Joseph e Pandya**.

Si considera, per ogni processo, il **Response Time** ( $R_i$ ), ossia il worst case time richiesto dal task  $i$ -esimo per completare la propria esecuzione.

Si considera la seguente formula matematica:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \quad (4.3)$$

dove

- $C_i$ : costo computazionale del task  $i$  su cui sta conducendo il calcolo del rispettivo  $R_i$
- $hp(i)$ : task più prioritari rispetto al task  $i$
- $T_j$  e  $D_j$ : rispettivamente periodo e deadline del task  $j$
- $\lceil x \rceil$ : funzione soglia (o ceiling) che ritorna il minimo intero maggiore o uguale. (Es.  $\lceil 5.2 \rceil = 6$ ).

L'analisi si basa sull'iterazione della formula (4.3) finché non si raggiunge la convergenza, ovvero:

$$R_i^{n+1} = R_i^n$$

L'inizializzazione del seguente algoritmo è:

$$R_i^0 = 0$$

Infine, il valore di  $R_i$  trovato si deve confrontare con la deadline dello stesso task; se vale che:

$$R_i \leq D_i$$

allora il tutto risulterà schedulabile.

Considerando le seguenti priorità statiche

$$\pi(P_1) \geq \pi(P_2) \geq \pi(P_3) \geq \pi(P_4)$$

Si calcolano i seguenti tempi di risposta:

- Per il task  $P_1$

$$R_1 = C_1 = 2$$

- Per il task  $P_2$ :

$$\begin{aligned} R_2^1 &= 2 + \left\lceil \frac{2}{6} \right\rceil 2 = 2 + (1 \cdot 2) = 4 \\ R_2^2 &= 2 + \left\lceil \frac{4}{6} \right\rceil 2 = 2 + (1 \cdot 2) = 4 \end{aligned}$$

Convergenza raggiunta.

- Per il task  $P_3$ :

$$\begin{aligned} R_3^0 &= C_3 = 8 \\ R_3^1 &= 8 + \left\lceil \frac{8}{6} \right\rceil 2 + \left\lceil \frac{8}{12} \right\rceil 2 = 8 + 4 + 2 = 14 \\ R_3^2 &= 8 + \left\lceil \frac{14}{6} \right\rceil 2 + \left\lceil \frac{14}{12} \right\rceil 2 = 8 + 6 + 4 = 18 \\ R_3^3 &= 8 + \left\lceil \frac{18}{6} \right\rceil 2 + \left\lceil \frac{18}{12} \right\rceil 2 = 8 + 6 + 4 = 18 \end{aligned}$$

Convergenza raggiunta.

Quinti a convergenza con tutti i tempi di risposta, si effettuano le verifiche:

$$\begin{cases} R_1 \leq D_1 \quad 2 \leq 6 & (OK) \\ R_2 \leq D_2 \quad 4 \leq 12 & (OK) \\ R_3 \leq D_3 \quad 18 \leq 24 & (OK) \end{cases}$$

Essendo valide tutte le diseguaglianze, l'analisi esatta di Joseph e Pandya dimostra che il task set è **schedulabile** nonostante il test di Liu & Layland sia fallito.

## Applicazione

Una simulazione dello scheduling è mostrata di seguito nella Figura (4.3).

All'istante iniziale i 3 processi arrivano; la CPU esegue prima  $P_1$ , poi  $P_2$  e infine  $P_3$ .

Al tempo  $t = 6$  è in esecuzione il processo  $P_3$  ma arriva nuovamente  $P_1$ . Avendo priorità maggiore, si interrompe l'esecuzione di  $P_3$  e verrà eseguito  $P_1$ .

Finita l'esecuzione di  $P_1$ ,  $P_3$  verrà eseguito e terminato.

All'istante  $t = 12$ , contemporaneamente sono disponibili  $P_1$  e  $P_2$ ; la CPU esegue prima  $P_1$  e poi  $P_2$ . Finita l'esecuzione,  $P_2$  verrà eseguito e terminato.

Come si può notare in figura, le frecce nere corrispondono all'arrivo del task rispettivo.

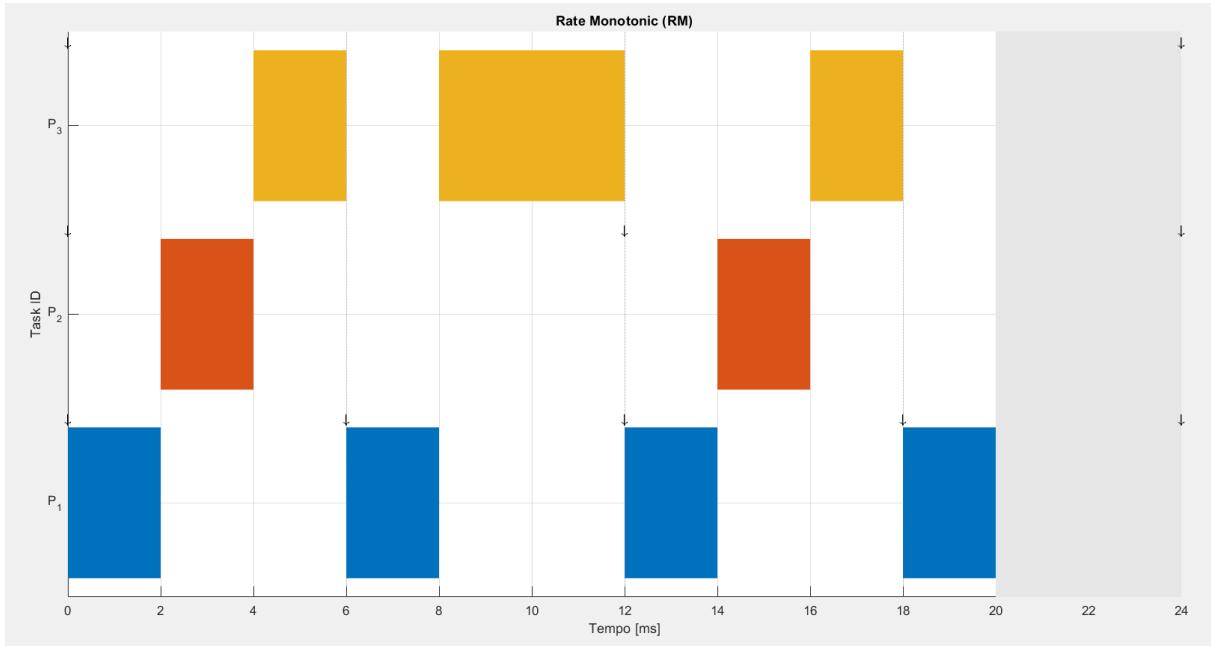


Figura 4.3: Rate Monotonic (MATLAB)

#### 4.2.4 Earliest Deadline First (EDF)

**Earliest Deadline First (EDF)** è un algoritmo di scheduling a **priorità dinamica**. In questo algoritmo la priorità viene assegnata e calcolata a runtime.

In ogni istante, tra tutti i processi pronti per l'esecuzione, lo scheduler seleziona quello che ha la deadline assoluta più imminente; quindi il processo con la deadline più vicina ottiene la priorità massima.

Si parla quindi di:

- **Deadline Relativa**: intervallo di tempo entro cui il task deve terminare rispetto al suo istante di attivazione
- **Deadline Assoluta**: istante di tempo reale entro cui il task deve finire. Si calcola sommando il tempo corrente di attivazione alla deadline relativa.

Anche l'algoritmo EDF è **pre-emptive**.

#### Applicazione

Come passo preliminare si effettua la seguente verifica:

$$U \leq 1$$

Avendo garantito in precedenza la seguente disuguaglianza, vale che, quest'ultima, sarà **condizione necessaria e sufficiente** affinché lo schedule sia fattibile.

Pertanto, l'algoritmo EDF si configura come **ottimale**, ovvero è in grado di generare uno schedule fattibile se questo esiste.

Si vuole effettuare una breve descrizione specifica del grafico (4.4):

- $t = 0 \rightarrow$  tutti i processi si attivano:

1. Confronto:  $D_1 : 6, D_2 : 12, D_3 : 24$ .

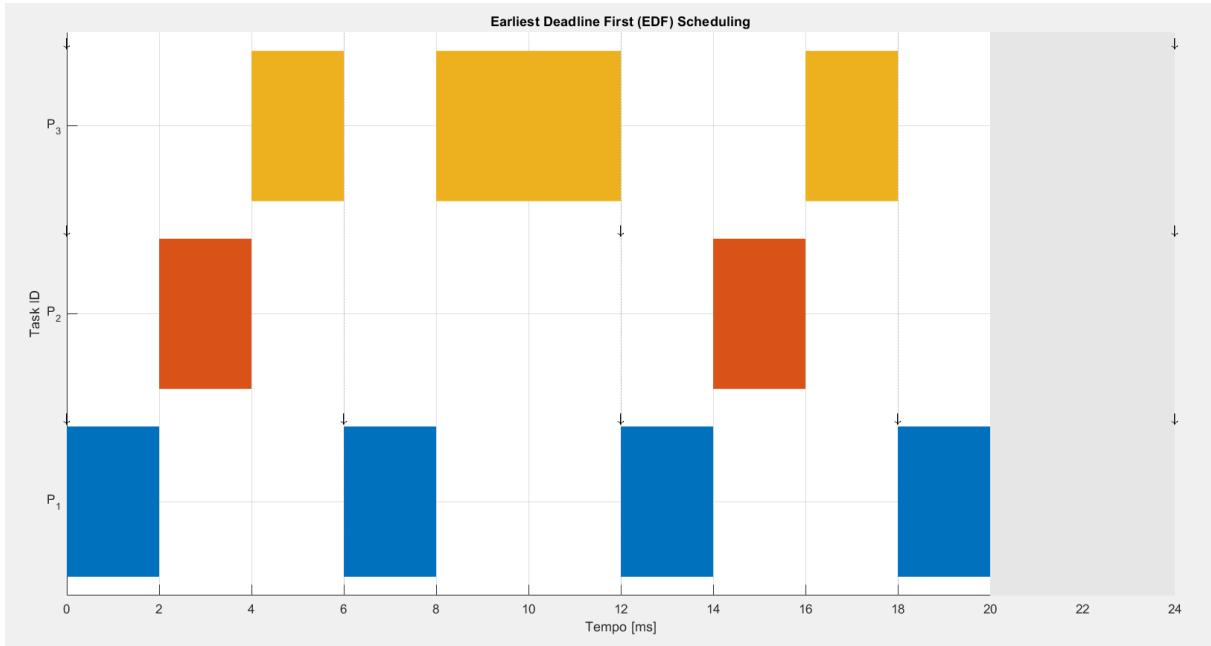


Figura 4.4: Earliest Deadline First (MATLAB)

2. ”Vince”  $P_1$ .

- $t = 2 \rightarrow P_1$  termina:

1. Confronto:  $D_2 : 12, D_3 : 24$ .

2. ”Vince”  $P_2$ .

- $t = 4 \rightarrow P_2$  termina:

1. ”Vince”  $P_3$  poiché è l’unico attivo.

- $t = 6 \rightarrow P_3$  è in esecuzione ma arriva  $P_1$ :

1. Nuova deadline assoluta per  $P_1$  :  $D_1 = 6 + 6 = 12$

2. Confronto:  $D_1 : 12, P_3 : 24$ .

3. ”Vince”  $P_1$ .

Si interrompe  $P_3$  e viene eseguito  $P_1$ .

- $t = 8 \rightarrow P_1$  termina e riprende la sua esecuzione  $P_3$ .

- $t = 12 \rightarrow P_3$  in esecuzione e arrivano  $P_1$  e  $P_2$  :

1. Nuova deadline assoluta per  $P_1$  :  $D_1 = 12 + 6 = 18$  e per  $P_2$  :  $D_2 = 12 + 12 = 24$

2. Confronto:  $D_1 : 18, D_2 : 24, D_3 : 24$ .

3. ”Vince”  $P_1$

Si interrompe  $P_3$  e viene eseguito  $P_1$ .

- $t = 14 \rightarrow P_1$  termina:

1. Confronto:  $D_2 : 24, D_3 = 24$
  2. Il processore sceglie arbitrariamente: "Vince"  $P_2$
- $t = 14 \rightarrow P_2$  termina:
    1.  $P_3$  finisce la sua esecuzione

#### 4.2.5 Least Slack Time (LST)

Nell'algoritmo **Least Slack Time** (LST), la priorità viene determinata dalla "urgenza" istantanea del processo.

Si parla di **slack**, il quale rappresenta il tempo "libero" che rimane ad un processo prima che diventi impossibile completarlo entro la sua scadenza.

La formula matematica che serve a calcolare lo slack ( $st$ ) al tempo corrente ( $t$ ) è:

$$st = (D - t) - C$$

dove  $D$  è la **deadline assoluta** del processo,  $t$  è il **tempo reale** corrente e  $C$  è la **porzione di computazione** non ancora eseguita (tempo residuo).

La logica di funzionamento è la seguente:

- Il processo che possiede il minimo tempo di slack ottiene la massima priorità.
- Se un processo in attesa vede il suo slack scendere al di sotto di quello del processo in esecuzione, avviene in interruzione (pre-emption).

#### Applicazione

Si considera il seguente scheduling:

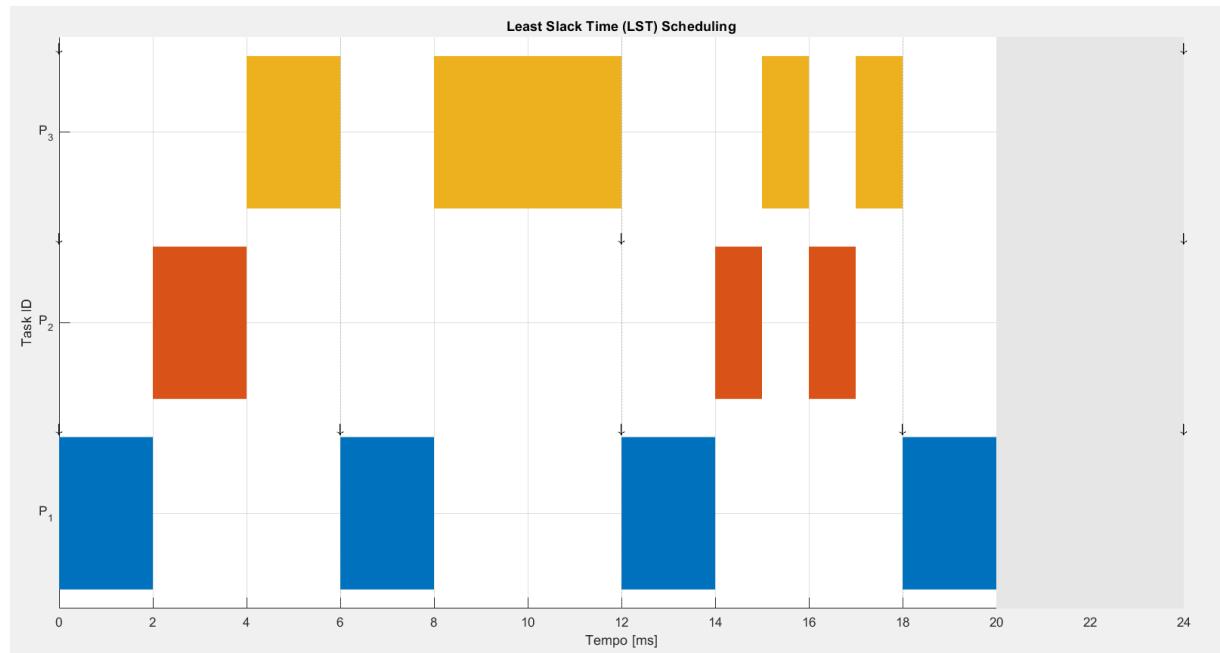


Figura 4.5: Least Slack Time (MATLAB)

Inizialmente, le scadenze sono lontane e i tempi di calcolo brevi, quindi gli slack sono ampi e distinti.

$P_1$  ha sempre lo slack minore appena viene rilasciato e quindi viene eseguito subito poiché possiede priorità massima.

La situazione "critica" si verifica a  $t = 14$ . I processi attivi sono  $P_2$  e  $P_3$  con le loro deadline assolute  $D_2 = 12 + 12 = 24$  poiché è la sua seconda attivazione  $D_3 = 24$  poiché è ancora attivo.

I loro tempi residui sono rispettivamente  $C_2 = 2$  poiché alla sua seconda attivazione non è ancora stato eseguito e  $C_3 = 2$ .

Calcolando gli slack:

$$\begin{aligned} st(P_2) &= (D_2 - t) - C_2 = (24 - 14) - 2 = 8 \\ st(P_3) &= (D_3 - t) - C_3 = (24 - 14) - 2 = 8 \end{aligned}$$

Avendo slack uguali, la CPU decide arbitrariamente; in questo caso viene eseguito  $P_2$ .

All'istante successivo  $t = 15$  si calcolano nuovamente gli slack:

$$\begin{aligned} st(P_2) &= (24 - 15) - 1 = 8 \\ st(P_3) &= (24 - 15) - 2 = 7 \end{aligned}$$

Poiché  $st(P_3) < st(P_2)$  si interrompe l'esecuzione di  $P_2$  e la CPU passa a  $P_3$ .

All'istante successivo  $t = 16$  gli slack risulteranno di nuovo uguali e qui, nuovamente la CPU sceglie arbitrariamente.

# Capitolo 5

## Conclusioni

I risultati ottenuti confermano l'efficacia degli strumenti utilizzati nel garantire il rispetto dei requisiti di sicurezza e temporalità.

Per quanto riguarda le **Reti di Petri**, l'analisi della *Catena di Montaggio* mediante il tool PIPE ha confermato la correttezza strutturale del modello. L'analisi dello spazio degli stati ha verificato la proprietà di Limitatezza (Boundedness), accertando che in nessun posto il numero di token superi il valore di 5 (corrispondente alle 5 materie prime in ingresso).

È stata inoltre dimostrata l'assenza totale di Deadlock, confermando che il sistema è in grado di resettarsi ciclicamente riabilitando il nastro trasportatore iniziale una volta completato l'assemblaggio finale.

Nella seconda parte, l'attenzione si è spostata sulla gestione dei vincoli temporali tramite i **Timed Automata**. Il caso di studio sulla *Gestione Energetica di un Robot Mobile* ha evidenziato l'importanza della sincronizzazione tra componenti paralleli.

Attraverso il model checker UPPAAL, le query di verifica hanno dimostrato la Safety del sistema (il robot non esaurisce mai la batteria andando in Fault) e la Liveness, garantendo che il ciclo di lavoro e ricarica possa proseguire indefinitamente rispettando i vincoli temporali stringenti.

Infine, l'analisi dello **Scheduling Real-Time** applicata a una *Centralina di Controllo (ECU)* automobilistica ha permesso di confrontare diverse strategie di allocazione della CPU per task con scadenze rigide (Hard Real-Time). I risultati hanno mostrato che:

1. Lo Static Cyclic Scheduling (SCS) offre massima predicitività ma richiede un'attenta scelta del major e minor cycle per minimizzare l'overhead.
2. L'algoritmo Rate Monotonic (RM), pur fallendo inizialmente il test sufficiente di Liu & Layland, si è dimostrato efficace tramite l'analisi esatta dei tempi di risposta, garantendo il rispetto di tutte le deadline.
3. L'algoritmo Earliest Deadline First (EDF) si è confermato ottimale per la gestione dinamica, mentre il Least Slack Time (LST), pur efficace, ha evidenziato il rischio di frequenti cambi di contesto quando i tempi di slack di processi concorrenti si equivalgono.

# Bibliografia

- [1] [http://www.scholarpedia.org/article/Petri\\_net](http://www.scholarpedia.org/article/Petri_net)
- [2] Slide del Corso di Programmazione di Sistemi Tempo-Reali e Distribuiti, *Introduzione agli Automi a Stati Finiti e alle Reti di Petri*, 2025
- [3] Gerd Behrmann, Alexandre David, and Kim G. Larsen, *A Tutorial on Uppaal*, 2004
- [4] Slide del Corso di Programmazione di Sistemi Tempo-Reali e Distribuiti, *Introduzione ai Timed Automata e Uppaal*, 2025
- [5] Slide del Corso di Programmazione di Sistemi Tempo-Reali e Distribuiti, *Real Time Scheduling*, 2025