



Politecnico di Bari

Dipartimento di Ingegneria Elettrica e dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione

Tesi in Controllo Digitale

**Realizzazione ed implementazione di un algoritmo di
controllo di semafori per la gestione ottimizzata del traffico
in un modello a più incroci**

Relatore:

Prof. Paolo Lino

Studente:

Giorgio Mannarini

Anno Accademico 2019 - 2020

Indice

Introduzione	5
1 Modellazione e gestione statica del singolo incrocio	7
1.1 Generazione delle automobili	9
1.2 Il modello in Simulink e SimEvents: introduzione	17
1.3 Il modello in Simulink e SimEvents: analisi dettagliata dei componenti	20
1.4 L'algoritmo di gestione statica dell'incrocio	28
1.5 Analisi dei risultati	30
1.6 Tabella riassuntiva dei risultati con valori variabili del parametro μ .	32
2 Algoritmo di gestione dinamica applicato ad un singolo incrocio	34
2.1 Breve spiegazione dell'algoritmo originale	36
2.2 Le modifiche apportate	38
2.3 Spiegazione del codice implementato	44
2.4 Il modello per confrontare le diverse gestioni del singolo incrocio . . .	48
2.5 I risultati ottenuti	51

Elenco delle figure

1.1	Direzioni compatibili	8
1.2	Accesso al workspace, ingrandimento	12
1.3	Inserimento di variabili globali e codice di inizializzazione nel modello	13
1.4	Vetture complessive nell'arco di 24 ore per una singola corsia	14
1.5	Modello di un incrocio a raso a quattro bracci in Simulink e SimEvents, gestione statica	17
1.6	Ingrandimento su una singola corsia	19
1.7	Entity Generator e relativa configurazione	20
1.8	Entity Queue e relativa configurazione	21
1.9	Entity Gate e relativa configurazione	22
1.10	Entity Server e relativa configurazione	23
1.11	Entity Terminator	24
1.12	Blocchi inerenti alla funzione di controllo	25
1.13	Il blocco Message Send	25
1.14	Il blocco timer realizzato	26
1.15	Il blocco Unit Delay	26
1.16	Il blocco Simulink Function per la funzione getTime()	27
1.17	Numero di auto in coda per una singola corsia	31
1.18	Tempi di attesa medi per una singola corsia in funzione dell'orario considerato	31
2.1	Modello di un incrocio a raso a quattro bracci in Simulink e SimEvents, gestione dinamica	40
2.2	Modello per comparare gli algoritmi utilizzati per la gestione di un singolo incrocio	50

Elenco dei codici

1.1	Entity Generator	11
1.2	Tassi di generazione variabili	12
1.3	Codice relativo ai parametri globali dell'Entity Server	23
1.4	Codice inserito negli Entity Server	23
1.5	Algoritmo di gestione statica del singolo incrocio	28
2.1	Pseudocodice di gestione dinamica di un incrocio	35
2.2	Implementazione dell'algoritmo di gestione dinamica di un singolo incrocio	41
2.3	Scelta delle due strade complementari a quella selezionata come principale	45
2.4	Porzione di codice relativa alla gestione della starvation	48

Elenco delle tabelle

1.1	Relazione fra il parametro μ ed i tempi di interarrivo	16
1.2	Esecuzioni dell'algoritmo di gestione statica al variare di μ	33
2.1	Tabella di comparazione fra algoritmi di gestione del singolo incrocio - prime quattro corsie	52
2.2	Tabella di comparazione fra algoritmi di gestione del singolo incrocio - ultime quattro corsie	53

Introduzione

Il controllo del traffico automobilistico mediante algoritmi performanti, soprattutto per quanto concerne le città più popolose, è un problema affrontato in numerosi documenti di stampo scientifico; tuttavia pochi sono stati i risultati finora raggiunti, ed il sistema più utilizzato per gestire il flusso di veicoli nelle intersezioni stradali resta quello dei semafori staticamente programmati, indipendenti cioè da parametri o eventi esterni.

Certamente negli anni si è compresa quantomeno l'importanza di assegnare una differente priorità, comunque statica, alle varie strade, in modo tale da privilegiare le arterie principali con meccanismi come le onde verdi, tuttavia sarebbe più auspicabile un processo a priorità dinamica, che conceda il verde alle strade più congestionate. Numerose sono le ricerche effettuate in tal senso, sia per quanto concerne l'algoritmo da adottare, che per quanto riguarda la sua relativa implementazione.

In questo lavoro di tesi si è selezionato uno degli algoritmi più comuni, lo si è modificato per migliorarne ulteriormente le performance, e lo si è applicato ad uno scenario di simulazione reale, anch'esso realizzato in maniera autonoma. È da chiarire che l'obiettivo non è stato quello di snellire il traffico quando tutte le arterie che convergono nell'incrocio sono sature di veicoli, quanto più di ottenere buone performance in questa situazione, migliorando molto però il comportamento dei semafori in condizioni di squilibrio, ovvero quando le strade affollate sono solo alcune delle confluenti.

Nello specifico, in primo luogo è stato necessario stabilire la tipologia di incrocio da analizzare, e in tal senso è stata scelta una intersezione a raso a quattro bracci [1], in cui ogni direzione è percorribile sia in un senso che in quello opposto, e nessuna svolta è vietata. A tal proposito, si è reso fondamentale creare il modello che simula il singolo incrocio in Matlab, Simulink e SimEvents, per monitorare l'andamento del traffico variando alcuni parametri come il tasso di arrivo dei veicoli e la durata del verde, iniziando da un classico algoritmo di gestione statica, implementato comunemente in questo tipo di giunzioni.

Il passo successivo è stato quello di stabilire quali sono le direzioni fra loro compatibili, alle quali cioè è possibile concedere il verde contemporaneamente senza causare

incidenti. È da sottolineare che questo lavoro viene normalmente svolto durante il posizionamento e la regolazione di un semaforo classico, e che per ogni tipo di incrocio esistono tabelle indicanti proprio tutte queste direzioni compatibili. La creazione di una tabella di questo genere è il primo passo per la definizione di un algoritmo di gestione ottimizzata dell'incrocio stesso.

Per quanto concerne l'algoritmo ideato, esso si basa su una proposta di *Maram Bani Younes* e *Azzedine Boukerche* [2], modificata ed opportunamente adattata. Si è proceduto dapprima a tradurre l'idea in codice, applicandola ad un singolo incrocio, sfruttando il modello realizzato in precedenza, per poi proseguire con la realizzazione di un modello più complesso e a più ampio respiro, che coinvolge nove diverse intersezioni, collegate fra loro, per simulare una situazione in cui le automobili che lasciano una giunzione confluiscono nella successiva.

A tal proposito, si è reso opportuno modificare ancora una volta l'algoritmo, per adattarlo ad una situazione di questo genere, ed i dati ricavati sono stati confrontati con quelli provenienti dallo stesso modello a nove incroci, gestito però con una turnazione classica dei semafori.

Nel **capitolo 1** viene dunque introdotto il primo modello realizzato, quello del singolo incrocio, e ne vengono analizzate le performance al variare del tasso di generazione delle auto e alla durata del verde per ciascuna strada, senza l'utilizzo di un algoritmo a priorità dinamica.

Nel **capitolo 2** viene esplicato il funzionamento dell'algoritmo sopracitato, le modifiche apportate, e viene mostrato come la sua applicazione al modello precedente, a parità di altre condizioni, migliori le performance dell'incrocio, misurate in termini di numero di auto in coda e tempo di attesa medio delle stesse.

Nel **capitolo 3** viene presentato il secondo modello realizzato, relativo ad una situazione con nove incroci interconnessi, e ne vengono analizzate le prestazioni in assenza di un algoritmo di controllo.

Nel **capitolo 4**, in fine, si analizza il comportamento di tale modello in presenza di un algoritmo che tenga conto del numero di auto in coda in ciascuna corsia, e di altri parametri esterni, confrontando i risultati ottenuti con quelli del capitolo precedente.

Capitolo 1

Modellazione e gestione statica del singolo incrocio

Come accennato nell'introduzione, è stato scelto di analizzare un incrocio a raso a quattro bracci, senza limitazioni in termini di svolte e direzioni percorribili. Questo significa che la giunzione, composta appunto da quattro strade disposte nella classica forma a croce, presenta quattro corsie per ogni strada. Per comodità e per adottare una convenzione di linguaggio, ci si riferirà alle automobili che non hanno ancora superato l'incrocio definendole “entranti”, chiamando invece “uscenti” le vetture che già sono passate attraverso la giunzione.

Le quattro corsie, dunque, sono così assegnate: due sono dedicate a veicoli uscenti, provenienti cioè da una delle altre tre strade dell'incrocio, mentre le altre due sono dedicate alle automobili entranti. Nello specifico, adottando una convenzione di “guida a destra”, comune nella maggior parte dei paesi europei, la corsia più a destra delle due entranti è riservata a chi vuole svoltare a destra o andare dritto, mentre quella più a sinistra a coloro i quali vogliono svoltare a sinistra.

Questa suddivisione è molto comune negli incroci italiani, ed è la soluzione più utilizzata nella assegnazione dei sensi di marcia in incroci di questo genere.

Le corsie a cui può essere concesso il verde contemporaneamente sono sempre e solo due, e per ogni corsia ne esistono due possibili complementari, che possono cioè essere scelte per ottenere il verde assieme alla corsia suddetta. La figura in basso è riassuntiva di questo modello, che è applicabile alla tipologia di incrocio scelto ed è indipendente dal tipo di algoritmo utilizzato per controllare i semafori.

Nella gestione ordinaria degli incroci stradali solo una delle due alternative è scelta, in maniera statica, per ogni corsia, in base al flusso di traffico che si prevede per ciascun senso di marcia.

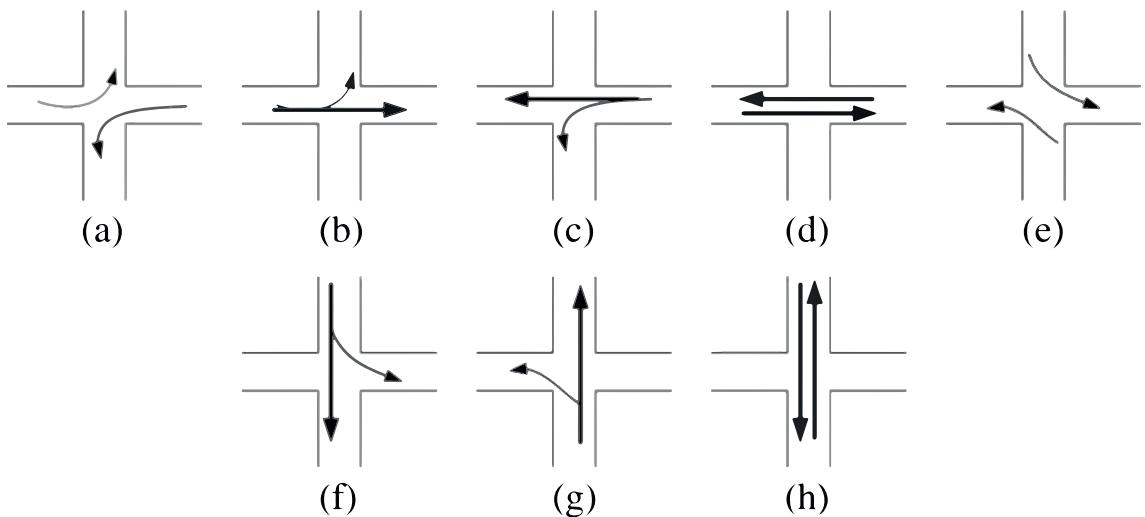


Figura 1.1: Direzioni compatibili

Nel modello oggetto di questo primo capitolo, è stato scelto di concedere il verde alle due corsie facenti parte della stessa strada, quella per svoltare a sinistra e quella per svoltare a destra / andare dritto, ovviamente a turno, per ognuno dei quattro bracci dell'incrocio (schemi (b), (c), (f), (g)). Ciò tuttavia non esclude che, essendo il sopracitato modello profondamente personalizzabile e parametrizzato, si possa prevedere anche la seconda opzione, indicata dagli schemi (a), (d), (e), (h) in figura.

1.1 Generazione delle automobili

Per quanto concerne la generazione delle auto, questa è indipendente per ogni corsia ed avviene mediante un processo di Poisson, utilizzato nella maggior parte dei modelli di simulazione del traffico [3]. Se di tale processo stocastico si è interessati alla probabilità che il tempo trascorso tra due eventi consecutivi sia pari a t è necessario fare ricorso ad una distribuzione esponenziale, di parametro λ , la cui funzione densità di probabilità $f(t)$ in questione, nello specifico, è la seguente.

$$PDF : f(t) = \lambda e^{-\lambda t} \quad (1.1)$$

Gli Entity Generator di Simulink, gli oggetti utilizzati per generare le automobili, prevedono tuttavia che la funzione che li governa sia riferita a dt , ovvero all'intervallo di tempo fra due eventi consecutivi, che deve essere una variabile casuale, e non alla sua funzione densità di probabilità. Pertanto, per ricavare tale intervallo di tempo dalla funzione densità di probabilità in oggetto, si fa riferimento alla sua funzione di distribuzione $F(t)$, ottenuta integrando $f(t)$ nell'intervallo $[0, t]$ e che rappresenta la probabilità che in tale intervallo si sia verificato un evento, in questo caso l'arrivo di un'automobile. Tale funzione di distribuzione ha la seguente espressione.

$$CDF : F(t) = 1 - e^{-\lambda t} \quad (1.2)$$

Da essa è facile ricavare la variabile t , che rappresenta l'incognita del problema, come inversa della CDF .

$$CDF^{-1} : t = -\frac{1}{\lambda} \ln [1 - F(t)] \quad (1.3)$$

Nello specifico $F(t)$, che, come già detto, è la probabilità che in un certo intervallo di tempo si verifichi un evento, è scelta casualmente mediante una distribuzione uniforme, fra 0 e 1. Pertanto, se $F(t) = 1$ si ha la certezza che c'è stato un arrivo, e questo può essere vero solo se $t \rightarrow +\infty$, al contrario per $F(t) = 0$ si è sicuri che non si sia verificato alcun evento, e ciò avviene solo se $t \rightarrow 0$.

Si nota, dunque, che più aumenta t , più è probabile che arrivi un'auto nella corsia in questione. Il tutto è governato dal parametro $\mu = \frac{1}{\lambda}$, arbitrariamente impostabile. Tale funzione è stata quindi implementata negli Entity Generator, che come già accennato si occupano di generare, appunto, entità, oggetti con determinate caratteristiche controllate dai due toolbox, Simulink e SimEvents.

Per rendere ancora più variabile e casuale il tempo medio che intercorre fra l'arrivo di un'auto e quello della successiva, il parametro μ , e dunque il tasso di generazione

delle vetture viene originato randomicamente ad ogni esecuzione, in un intervallo precedentemente determinato, utilizzando questa volta una distribuzione uniforme, capace dunque di assicurare assoluta equi-probabilità. Inoltre, poiché si prevede che durante le ore del giorno il numero di veicoli vari sensibilmente fra le ore di punta, le ore notturne e quelle intermedie, è stato anche previsto un meccanismo per permettere a SimEvents di rilevare in quale ora del giorno si trova durante la simulazione, e aumentare o diminuire di conseguenza μ . In definitiva, il codice utilizzato per la generazione delle automobili è alla seguente pagina.

```

1 %Random number generation
2
3 coder.extrinsic('rand');
4 ValEntry1 = 1;
5 ValEntry1 = rand();
6 %ValEntry1 rappresenta la CDF
7
8 index = 1;
9 %indice del generator, in questo caso il primo.
10
11 % Ora del giorno
12 time = mod(getTime(), 86400) / 3600;
13
14 % Fasce orarie
15 if time > 0 && time < 7
16 myMu = muNotte(index);
17 elseif time > 7.5 && time < 9
18 myMu = muMattinaPranzo(index);
19 elseif time > 12 && time < 14.5
20 myMu = muMattinaPranzo(index);
21 elseif time > 18 && time < 20
22 myMu = muSera(index);
23 else
24 myMu = muArray(index);
25 end
26
27 % Pattern: distribuzione esponenziale
28 dt = -myMu * log(1 - ValEntry1);
29 % dt: tempo fra due eventi di Poisson

```

Codice 1.1: Entity Generator

Come è facile comprendere, si ha, rispetto alla descrizione precedente, $\mu = myMu$. È inoltre da precisare che i parametri *muNotte*, *muMattinaPranzo*, *muSera* e *muArray* sono definiti come variabili globali, con questo codice.

```

1 coder.extrinsic('rand');
2 temp = 1;
3 temp = rand([1 8]);
4 %creo 8 numeri casuali fra 0 e 1, uno per ogni generator
5 muArray = 20 + (35 - 20) * temp;
6 %l'intervallo scelto va da 20 a 35
7
8 muNotte = muArray + 80; %di notte il tasso è molto basso
9 muMattinaPranzo = muArray - 6; %di mattina ed a pranzo è
   molto alto
10 muSera = muArray - 3; %di sera è più alto del normale

```

Codice 1.2: Tassi di generazione variabili

Per definire delle variabili globali è necessario accedere al *Model Workspace* mediante il pulsante in basso a sinistra in *figura 1.2*. Una volta fatto ciò, spostandosi sul modello (in questo caso *GateControl*), e sulla tab *Callbacks* in alto a destra, è possibile inserire il codice appena presentato cliccando su *InitFcn*, ovvero lo spazio in cui possono essere dichiarate le funzioni / variabili di inizializzazione del modello.

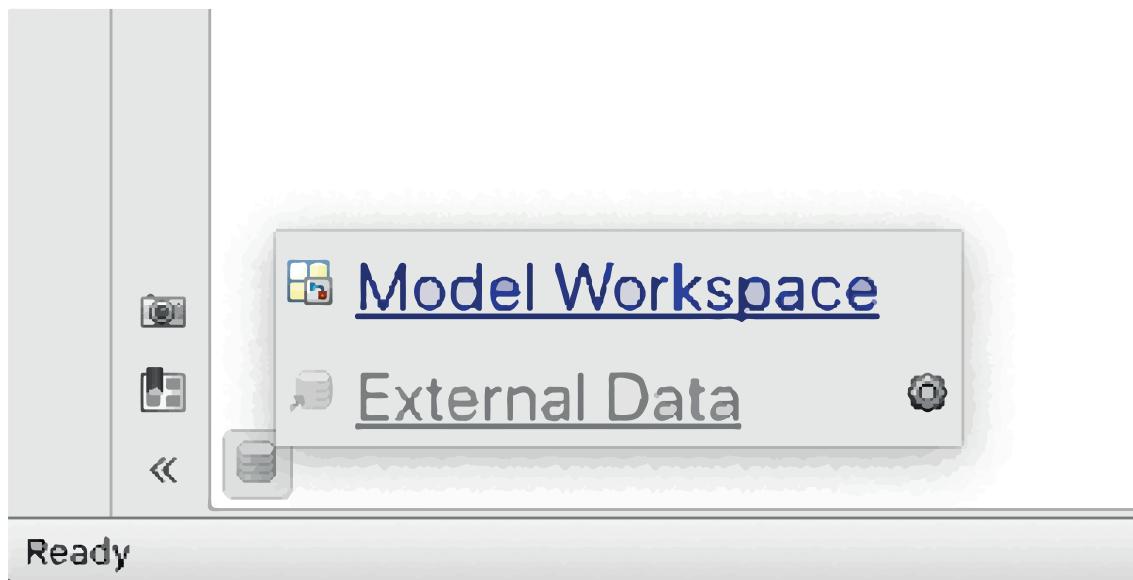


Figura 1.2: Accesso al workspace, ingrandimeto

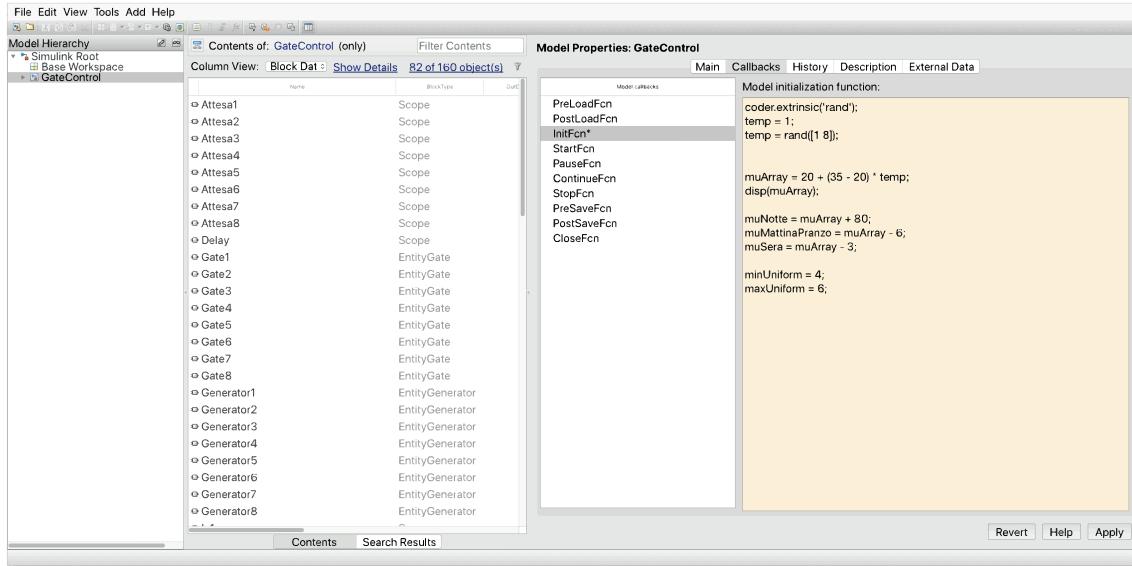


Figura 1.3: Inserimento di variabili globali e codice di inizializzazione nel modello

Si può notare dal *codice 1.2* che vengono generati otto numeri casuali, fra 20 e 35, che costituiscono i valori base per il parametro μ . Ciò vuol dire che mediamente fra l'arrivo di un'auto e della successiva intercorre un tempo compreso tra:

$$dt = -20 \ln [1 - 0.5] = 13.86s$$

e

$$dt = -35 \ln [1 - 0.5] = 24.26s$$

dove 0.5 è il valore medio della distribuzione uniforme precedentemente descritta. Tali valori base sono utilizzati nelle ore intermedie, come già detto, mentre sono modificati durante le ore di punta e quelle notturne.

Il valore di μ è poi profondamente decrementato durante le ore di punta (mattina e ora di pranzo), incrementato durante la notte e decrementato, anche se meno drasticamente, all'ora di cena, influendo ovviamente sui tempi di interarrivo, che diventano i seguenti.

Mattina e pranzo (fra le ore 7.30 e le ore 9.00, e fra le ore 12.00 e le ore 14.30): da 9.7 secondi a 20.1 secondi.

Sera (fra le ore 18.00 e le ore 20.00): da 11.78 secondi a 22.18 secondi.

Notte (fra le ore 0.00 e le ore 7.00): da 69.31 secondi a 79.71 secondi.

Il tutto è stato fatto per rendere il modello quanto più imprevedibile possibile, mantenendolo però fedele alla realtà.

Si nota che, con questi tassi, vengono mediamente generate 2200 automobili per ogni corsia, ovvero circa 17600 vetture attraversanti l'incrocio durante l'arco di 24 ore, come riportato dal seguente grafico.

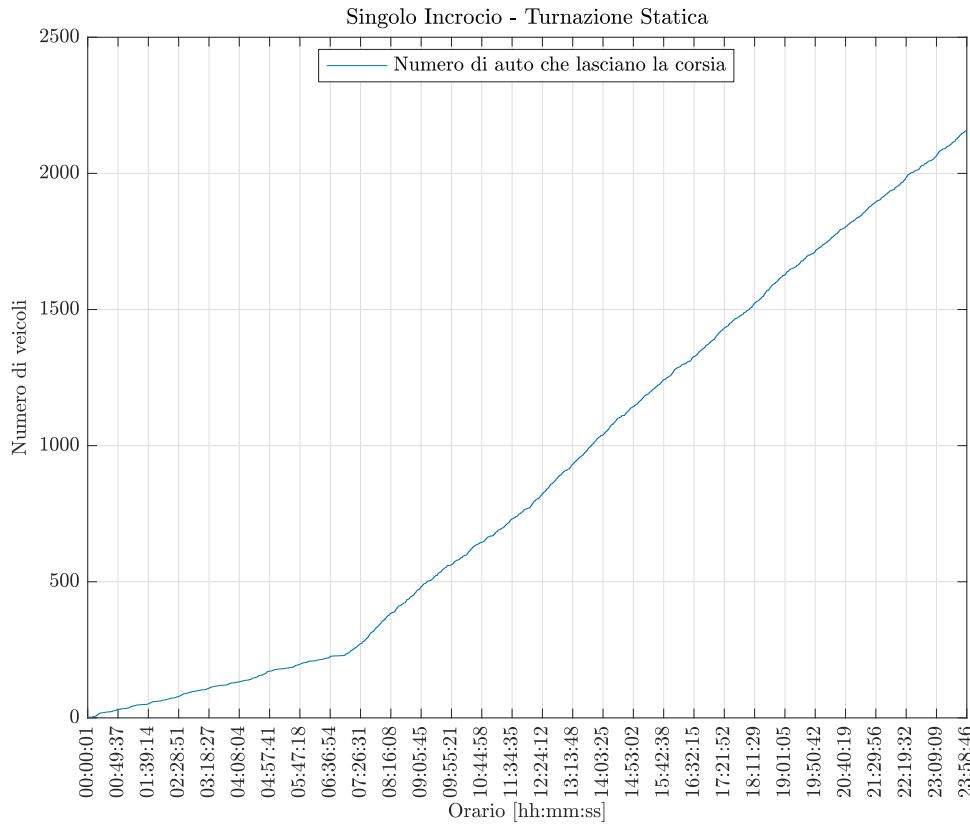


Figura 1.4: Vetture complessive nell'arco di 24 ore per una singola corsia

La scelta di questi valori non è stata casuale: si è voluto infatti considerare il caso di un'intersezione affollata, plausibile nelle grandi città. Variando il parametro manualmente e sperimentando rispetto al numero di automobili, si è trovato in questi numeri un buon compromesso per analizzare il comportamento del modello e dell'algoritmo applicato.

È infatti da notare che negli incroci in cui mediamente il numero di veicoli giornalieri è molto basso l'applicazione di un algoritmo di controllo come quello che sarà presentato nel capitolo successivo di questo lavoro di tesi risulta sì essere utile, ma meno drasticamente determinante, in quanto non vi è una situazione di traffico intenso e di profondo squilibrio, con strade molto affollate e strade con poche automobili, situazione che invece verrà ricreata, in seguito, per confrontare l'algoritmo statico di questo capitolo con quello dinamico, di cui si discuterà.

Anche le variazioni in base alle fasce orarie sono state analizzate, e non lasciate al caso: si è voluto infatti simulare un pesante congestramento nelle ore del giorno mediamente più critiche per una giunzione strategica di una città, ossia fra le ore 7.30 e le ore 9.00, quando gli alunni prendono posto in classe ed i lavoratori arrivano nei loro uffici, così come fra le ore 12.00 e le ore 14.30, e in serata, fra le ore 18.00 e le ore 20.00. Si è considerato, tuttavia, che in quest'ultima fascia oraria, complice l'assenza dei bambini, ormai a casa, e di molti anziani, il tasso sia sì più elevato rispetto a quello standard, per via dei lavoratori, di ritorno alle loro abitazioni, ma meno rispetto a quello mattutino e dell'ora di pranzo. Durante la notte, invece, si è voluto mantenere il numero di veicoli molto basso, anche questa volta facendo riferimento ad un grande incrocio cittadino.

Quindi in definitiva i valori di μ sono frutto di simulazioni sperimentali e di un compromesso fra una situazione realmente plausibile ed un congestramento dell'incrocio comunque elevato.

Per concludere, la seguente tabella è riassuntiva rispetto ai valori di μ scelti in base all'ora del giorno ed ai relativi tempi di inter-arrivo. Nel paragrafo conclusivo di questo capitolo, a tale tabella verranno aggiunti anche i tempi medi di attesa di ciascuna automobile nonché il numero medio di auto in coda per ciascuna corsia, e sarà certamente più chiara la corrispondenza fra il parametro ed il flusso di autovetture.

Fascia Oraria	Intervallo μ	Stima tempo minimo fra due arrivi	Stima tempo massimo fra due arrivi
0.00 - 7.00	100 - 115	69.31s	79.71s
7.00 - 7.30	20 - 35	13.86s	24.26s
7.30 - 9.00	14 - 29	9.7s	20.1s
9.00 - 12.00	20 - 35	13.86s	24.26s
12.00 - 14.30	14 - 29	9.7ss	20.1s
14.30 - 18.00	20 - 35	13.86s	24.26s
18.00 - 20.00	17 - 32	11.78s	22.18s
20.00 - 0.00	20 - 35	13.86s	24.26s

Tabella 1.1: Relazione fra il parametro μ ed i tempi di interarrivo

È importante anche indicare la presenza di lassi temporali intermedi, fra un cambiamento ed un altro, come quello relativo alla fascia oraria 7.00 – 7.30, in cui il valore di μ viene posto al valore standard, che rappresenta a tutti gli effetti una “via di mezzo” fra la stima precedente e la successiva. Questo per non provocare un cambiamento troppo repentino del tasso di arrivo dei veicoli, inverosimile nella realtà, in cui aumenta e diminuisce gradualmente.

1.2 Il modello in Simulink e SimEvents: introduzione

Per quanto concerne il resto del modello, come si può notare nella figura seguente, ogni strada è rappresentata, oltre che dall'Entity Generator relativo, da una coda di tipo FIFO (First In First Out), la cui capienza può essere variata a piacimento, da un Entity Gate, che rappresenta il semaforo, e che dunque può assumere lo stato di *chiuso* (semaforo rosso) o *aperto* (semaforo verde), da un Entity Server, utile per modellare il tempo che una singola macchina trascorre nell'incrocio vero e proprio, quindi da quando lascia la strada da cui proviene a quando imbocca quella di uscita, ed in fine da un Entity Terminator, che non fa altro che eliminare dal modello i veicoli che hanno già lasciato l'incrocio.

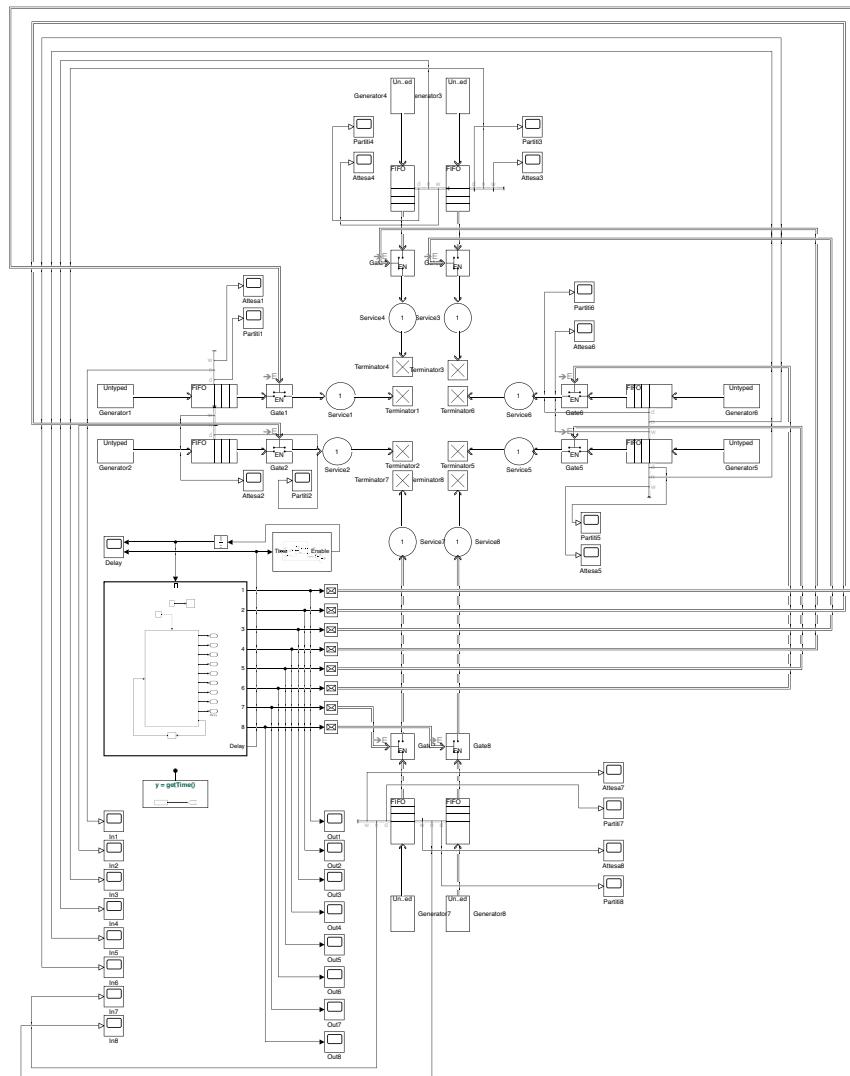


Figura 1.5: Modello di un incrocio a raso a quattro bracci in Simulink e SimEvents, gestione statica

Sono da fare alcune importanti precisazioni:

1. La grandezza delle code è stata impostata ad infinito, questo per valutare l'efficienza dei due algoritmi e non avere risultati falsati dati dal fatto che le code si riempiono. In altre parole, non ci sono limiti, in questo modello, a quante macchine possono essere presenti in una singola corsia, tuttavia questi limiti possono essere tranquillamente imposti per poter rendere lo schema più fedele alla realtà. Ciò vale esclusivamente per il modello a singolo incrocio, per i motivi suddetti. Nel Capitolo 3 verrà introdotto un nuovo modello, con più incroci interconnessi, e questo, per adattarsi meglio al funzionamento reale, sarà proposto con code che presentano una capienza massima, condizionata dalla distanza della giunzione in questione da quella precedente.
2. Il tempo che una macchina trascorre al centro dell'incrocio, definito all'interno degli Entity Server, ha una componente casuale, in quanto si è osservato che anche questo parametro, nel mondo reale, non è fisso, e si è voluta inserire un'ulteriore variabile di imprevedibilità nel modello. Nello specifico si è utilizzata una distribuzione uniforme in un intervallo [4, 6] secondi, come si può notare anche nel *codice 1.3* inerente alla spiegazione specifica del componente in questione. Questi valori non sono casuali: si è voluto stimare il tempo che una macchina impiega ad abbandonare l'incrocio. Considerando che la prima automobile della coda è la più lenta, mentre l'ultima sarà certamente la più veloce, è stato necessario stimare un valore medio. Tale stima resta confinata in un intervallo ristretto perché si è voluto mantenere un valore medio di 5s, senza peggiorare il valore limite superiore, essendo piuttosto inverosimile che un'automobile impieghi più di 6s da quando entra nella giunzione a quando ne esce, in assenza di cause esterne.
3. È da approfondire anche il concetto della luce gialla: il giallo, infatti, indica alle macchine che non hanno ancora superato il semaforo di fermarsi, e a quelle che stanno occupando l'incrocio di sgomberarlo al più presto. Questo, in effetti, è implicitamente implementato: quando l'Entity Gate si chiude, le macchine al centro dell'incrocio lo liberano, essendo già all'interno dell'Entity Server, consentendo poi a quelle provenienti dalle altre corsie di fluire.

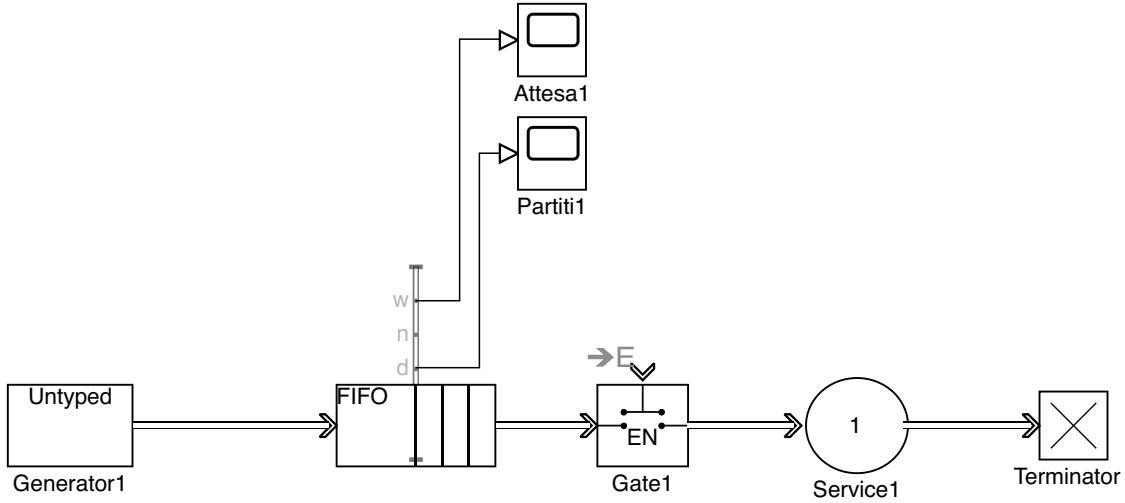


Figura 1.6: Ingrandimento su una singola corsia

Oltre ai numerosi oscilloscopi presenti, utili a visualizzare l’evoluzione nel tempo di una serie di informazioni, come il numero di macchine in ogni coda, la durata del verde per ciascun semaforo, il numero di macchine totali che hanno attraversato ogni corsia, il tempo medio d’attesa di ogni corsia, è da sottolineare che l’intero incrocio è governato da un blocco di tipo Enabled Subsystem, all’interno del quale è stata inserita una funzione Matlab, in un blocco Function-Call Subsystem. Ciò che questo blocco consente, è di eseguire tale funzione, una sola volta, solo quando viene inviato un segnale di trigger.

In altre parole, la funzione si occupa di cambiare lo stato del sistema (dare il rosso ai semafori attualmente verdi e concedere il verde ai successivi, per ora staticamente), e deve essere eseguita solo quando effettivamente il tempo concesso alla luce verde dalla chiamata precedente si esaurisce.

A tale scopo, il trigger è gestito da un timer, oggetto non presente in alcuna libreria, che si è prodotto in autonomia. Questo timer accetta in ingresso il tempo, in secondi, per il quale deve restituire in uscita 0, che equivale a *trigger inattivo*, capisce da SimEvents il numero di secondi passati dall’ultima chiamata, e ad ogni passo di campionamento (ogni secondo), esegue il confronto fra questo dato e quello in input, riuscendo così a richiamare la funzione ad intervalli temporali desiderati e ben precisi. Attualmente questo meccanismo sembra esageratamente complesso, in quanto i semafori hanno delle tempistiche prefissate, che saranno più chiare nel *paragrafo 1.4*, dedicato alla spiegazione dell’algoritmo di questo primo capitolo, tuttavia in previsione dell’implementazione di una politica di decisione dinamica, sia per quanto concerne le direzioni a cui assegnare il verde, che per la durata dello stesso, si è reso necessario adottare questo tipo di oggetto.

1.3 Il modello in Simulink e SimEvents: analisi dettagliata dei componenti

Come accennato in precedenza, la *figura 1.6* è riassuntiva di come è stata modellata una singola corsia. Nello specifico verrà proposta in questo paragrafo una spiegazione più dettagliata di ogni componente.

Entity Generator[4]: serve a generare entità. Le entità in Simulink e SimEvents sono oggetti con determinate caratteristiche, che possono muoversi all'interno del modello fra componenti collegati fra di loro. In questo caso le entità rappresentano le vetture.

Ogni entità è un oggetto a sé stante ed ha un proprio ciclo di vita: viene generata da un Entity Generator e terminata da un Entity Terminator, descritto in seguito. Per configurare un Entity Generator bisogna modificare, dopo aver aperto la finestra ad esso relativa con un doppio click, lo script relativo ad “Entity Generation”, in cui è stato inserito il *codice 1.2*. Il metodo di generazione è “Time-Based”, ovvero basato su una distribuzione che restituisce a Simulink un numero, rappresentante del tempo di inter-arrivo. In sostanza ad ogni generazione questa funzione viene richiamata, e restituisce l'intervallo di tempo che intercorrerà fino alla generazione successiva. Fondamentale è la spunta su “Generate entity at simulation start”, che fa partire il tutto. Tutti gli altri parametri sono stati lasciati invariati.

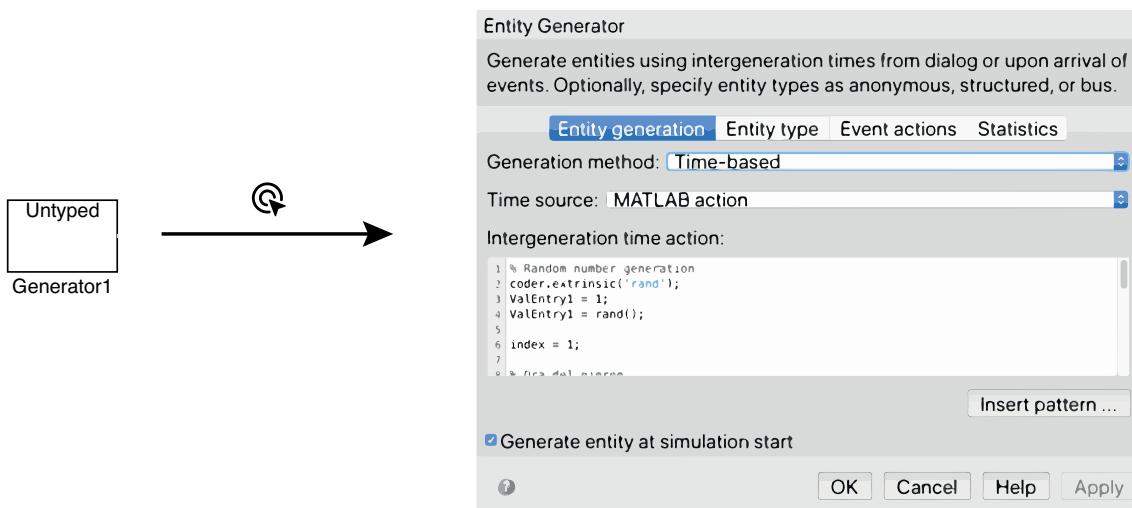


Figura 1.7: Entity Generator e relativa configurazione

Entity Queue[5]: il suo scopo è quello di implementare una coda di tipo First In First Out, all'interno della quale convergono le entità. Nel modello, rappresenta la corsia nella quale giungono le vetture, e può essere collegata direttamente all'Entity Generator, in quanto anche questo componente lavora con le entità. Una volta che un'automobile viene generata, essa automaticamente confluisce in questa coda. Anche questa volta, a seguito di un doppio click sono stati configurati alcuni parametri. Nello specifico “Capacity” è stato posto uguale ad una variabile globale, *queueSize*, che in questo modello è pari ad infinito per i motivi già spiegati in precedenza. È importante impostare il “Queue type” a FIFO, e comunicare alla coda che le entità giungeranno mediante la sua porta di ingresso (Entity arrival source pari a Input port). Per collegare oscilloscopi alla coda per visionare informazioni utili ci si può spostare nella tab “Statistics” e spuntare le caselle desiderate.

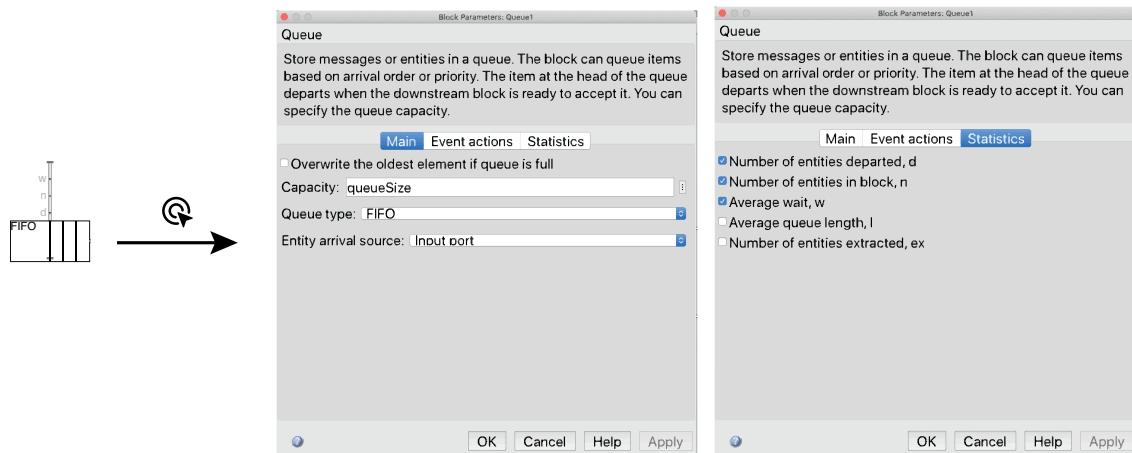


Figura 1.8: Entity Queue e relativa configurazione

Entity Gate[6]: rappresenta il semaforo. Quando è aperto, le entità sono libere di fluirvi attraverso, esattamente come un semaforo verde. È importante selezionare, nella finestra che si apre cliccando due volte sull'elemento, come modalità operativa “Enable gate”, in modo tale da farlo funzionare nel modo descritto, ovvero come chiuso quando in ingresso riceve uno 0 ed aperto quando riceve un 1. Si è scelto, inoltre, di partire da una situazione in cui tutti i semafori sono rossi, quando il modello viene inizializzato, quindi la casella “Open gate at simulation start” è stata deselezionata.

L’apertura e la chiusura del gate può essere controllata collegando un elemento di controllo alla porta in alto, che rappresenta il suo trigger. Tale elemento è un output del blocco Enabled Subsystem, di cui si parlerà poco più avanti.

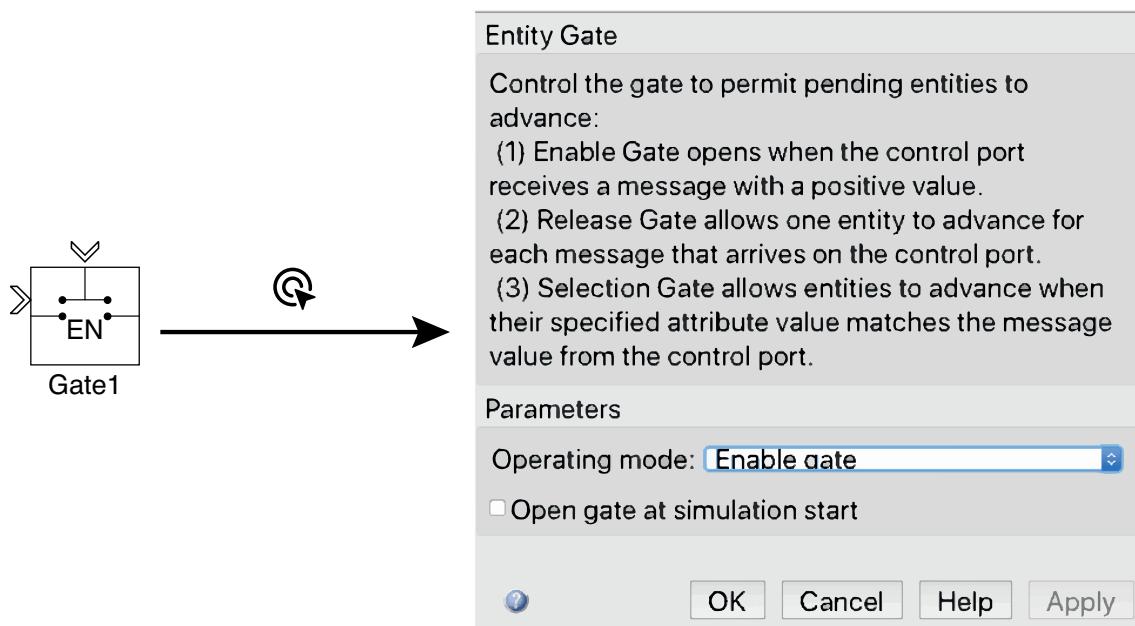


Figura 1.9: Entity Gate e relativa configurazione

Entity Server[7]: la sua funzione è quella di rappresentare l'incrocio vero e proprio, da quando una vettura supera il semaforo a quando entra nella corsia di uscita. Come spiegato, un'automobile impiega del tempo per percorrere tale tratto di strada, tempo che appunto viene modellato in questo componente. Con un doppio click sull'Entity Server, come al solito, è possibile modificarne i parametri. Nello specifico si è impostata la capacità di questo elemento ad 1, inserendo il seguente codice che modella la variabilità del tempo che un'automobile passa nell'incrocio. Le variabili *minUniform* e *maxUniform* sono globali, e sono proprio 4 e 6, come spiegato in precedenza e come si può notare nel *codice 1.3*. Tutte le altre impostazioni non sono state modificate.

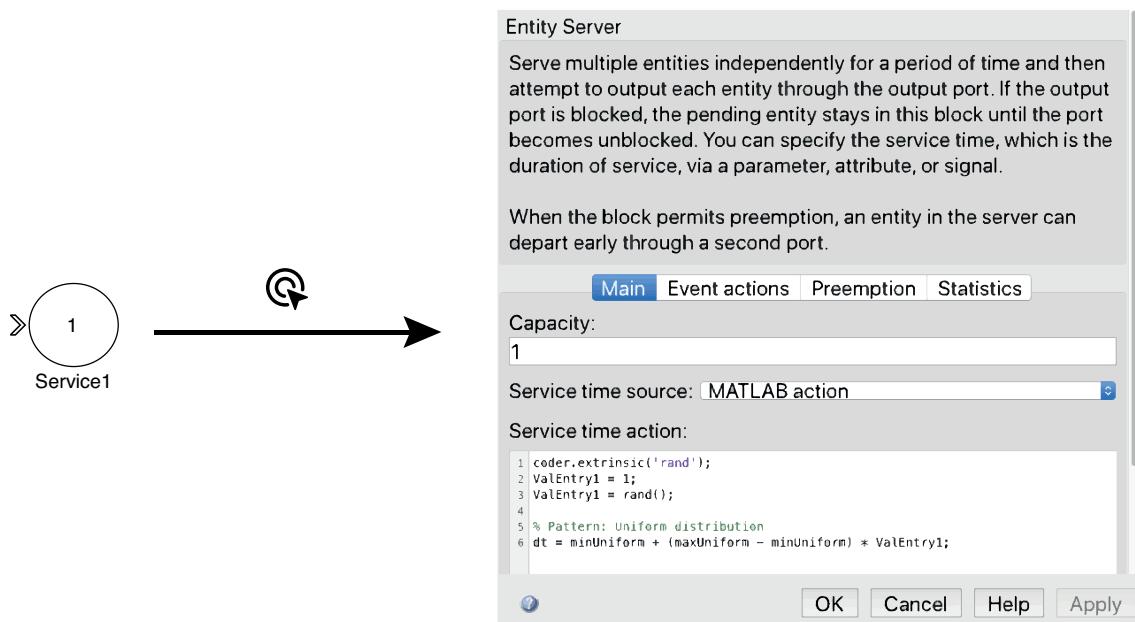


Figura 1.10: Entity Server e relativa configurazione

```
1 coder.extrinsic('rand');
2 minUniform = 4;
3 maxUniform = 6;
```

Codice 1.3: Codice relativo ai parametri globali dell'Entity Server

```
1 coder.extrinsic('rand');
2 ValEntry1 = 1;
3 ValEntry1 = rand();
4 % Pattern: Uniform distribution
5 dt = minUniform + (maxUniform - minUniform) * ValEntry1;
```

Codice 1.4: Codice inserito negli Entity Server

Entity Terminator[8]: serve ad eliminare le entità dal modello, viene collegato all’Entity Server e non è stato modificato in alcun modo.

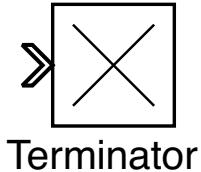


Figura 1.11: Entity Terminator

Enabled Subsystem[9]: è un blocco “container”, e serve a fare in modo che ciò che è presente al suo interno venga eseguito, una sola volta, quando alla porta di trigger di questo componente arriva un segnale pari ad 1. Viene collegato al timer realizzato, che sarà spiegato più avanti. Cliccandoci due volte è possibile notare che al suo interno è posto un blocco **Function-Call Subsystem[10]**, che contiene poi la funzione di controllo (Matlab) vera e propria.

Tutto questo è stato fatto per garantire che la logica di controllo sia effettivamente richiamata solo allo scadere del timer, ovvero quando l’incrocio deve passare da uno stato ad un altro. Il blocco Function Call Subsystem presenta in uscita gli output della funzione al suo interno, descritti nel paragrafo successivo. Questi output, poi, sono inviati al blocco Enabled Subsystem, che si fa carico di inviarli al modello principale. È importante notare che, poiché nel modello si ragiona con le entità, mentre le uscite di una funzione di Matlab sono delle variabili, bisogna incapsulare tali variabili in entità appunto. Ciò è possibile utilizzando dei blocchi di tipo **Message Send[11]**, che fanno da intermediari fra le uscite dell’Enabled Subsystem ed i componenti che tali uscite governano, ovvero gli Entity Gate, come sarà più chiaro, ancora una volta, nel prossimo paragrafo.

Come si può notare, uno degli output non è inviato al modello principale, ma è incapsulato in un blocco di tipo **Function-Call Feedback Latch[12]**. Questo output rappresenta lo stato dell’incrocio, che servirà da input alla chiamata successiva per concedere il verde ai semafori secondo una turnazione. È anche interessante notare che l’uscita *delay*, che non è altro che il tempo assegnato alla luce verde, non ha bisogno di essere incapsulata in un messaggio, essendo direttamente utilizzata dal timer.

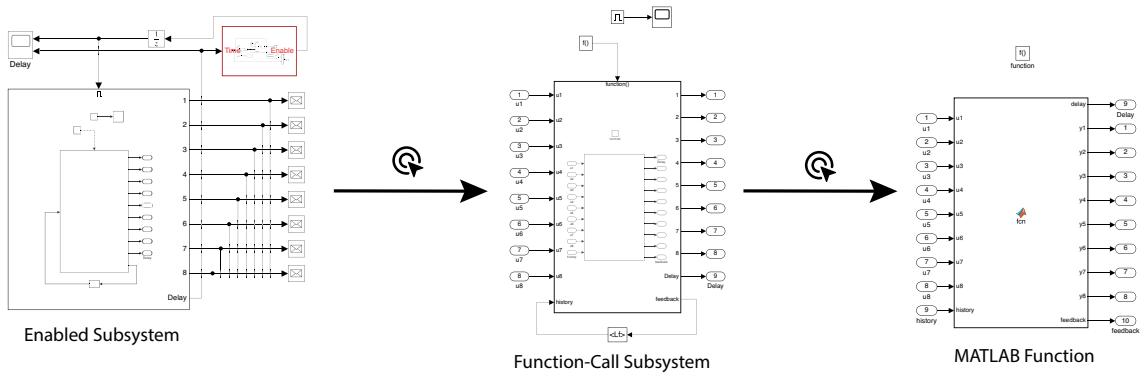


Figura 1.12: Blocchi inerenti alla funzione di controllo

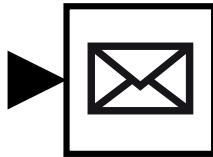


Figura 1.13: Il blocco Message Send

Il timer realizzato: anche questo componente è in realtà un sottosistema, ovvero, in un blocco **Subsystem[13]** è stato realizzato un meccanismo per inviare in uscita il numero 1 allo scadere di un timer. Prima di entrare nel merito di quanto creato, è bene notare che l'uscita non controlla direttamente il trigger del blocco Enabled Subsystem, in precedenza descritto, ma che prima venga ritardata di un secondo grazie a un blocco **Unit Delay[14]**. Ciò si è reso necessario per evitare alcuni errori, stabilizzando tale uscita.

È dunque chiaro che il timer in realtà non considera esattamente il tempo inviato in uscita dalla funzione, ma aggiunga ad esso un secondo preventivamente, secondo poi recuperato dall'attesa data da questo blocco.

Cliccando due volte sul sottosistema, si può notare come esso sia alquanto elementare. Sfruttando un contatore e fissando il passo di campionamento ad un secondo, ciò che succede è che ogni secondo tale contatore viene incrementato di 1, e viene confrontato con il valore in ingresso, a cui come precedentemente detto viene sommato 1 secondo. Quando questi due valori sono uguali, ovvero quando la loro differenza è nulla, allora il contatore viene resettato, e viene fornita un'uscita pari ad 1.

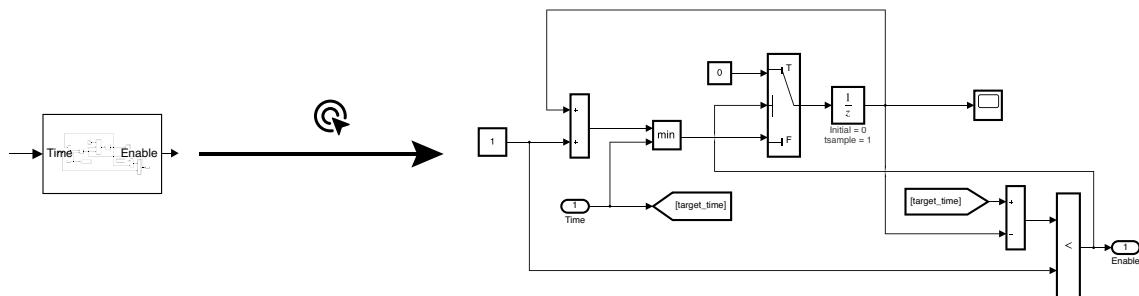


Figura 1.14: Il blocco timer realizzato

$$\frac{1}{z}$$

Figura 1.15: Il blocco Unit Delay

La funzione getTime: per poter acquisire l'orario della simulazione si è creato, anche questa volta, un oggetto apposito, incapsulato in un blocco **Simulink Function**[15]. Nello specifico, si è usato un **Digital Clock**[16], che restituisce un valore numerico che rappresenta il numero di secondi trascorsi dall'avvio della simulazione. Poiché è stato scelto di far durare tale simulazione 86400s, ovvero un giorno, questo numero oscilla fra lo 0 (corrispondente alla mezzanotte) ed 86400, appunto. Collegando tale blocco all'uscita della Simulink Function, si può richiamare la funzione ogni volta che lo si necessita, comprendendo così come far variare i tassi di generazione in base all'ora del giorno.

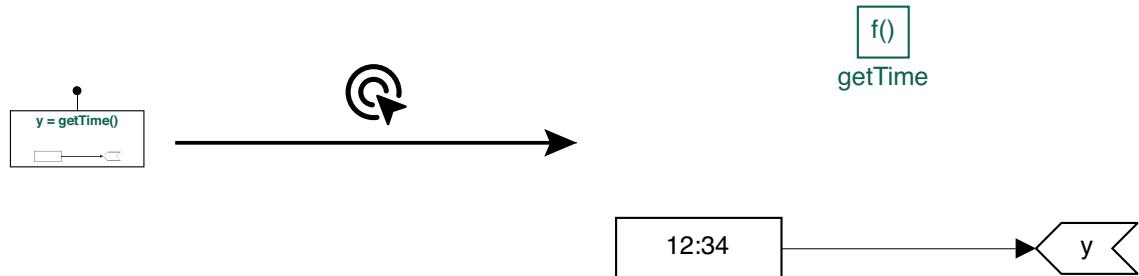


Figura 1.16: Il blocco Simulink Function per la funzione `getTime()`

1.4 L'algoritmo di gestione statica dell'incrocio

In questa prima fase, come precedentemente accennato, si è voluto modellare il comportamento di un incrocio di questo genere in maniera quanto più possibile fedele alla realtà, per avere un metro di paragone per le successive elaborazioni. Nello specifico, con riferimento alla *figura 1.1*, e come già detto, sono state scelte le direzioni (b), (c), (f), (g), da alternare, concedendo a ciascuna una durata del verde pari a 25 secondi, parametro plausibile se si considera un incrocio grande ed affollato, cosa effettivamente vera, viste le circa 17600 macchine giornaliere.

L'algoritmo implementato è il seguente.

```
1 function [ delay , y1 , y2 , y3 , y4 , y5 , y6 , y7 , y8 , current ] =
  fcn(previous)
2
3 GreenTime = 25;
4 % Current e previous indicano a quale strada si sta dando /
  si è dato il verde
5 current = zeros([1 4]);
6 next = 1; %Elemento del vettore a cui dare il verde
7 for i = 1 : 1 : 4
8 if previous(i) == 1
9 next = mod(i, 4) + 1;
10 end
11 end
12 current(next) = 1;
13 %Risultato
14 U = ([0 0; 0 0; 0 0; 0 0]);
15 U(next, 1) = 1;
16 U(next, 2) = 1;
17 delay = GreenTime; y1 = U(1, 1); y2 = U(1, 2); y3 = U(2, 1);
  y4 = U(2, 2); y5 = U(3, 1); y6 = U(3, 2); y7 = U(4, 1);
  y8 = U(4, 2);
18 %Assegnazione delle uscite
19
20 end
```

Codice 1.5: Algoritmo di gestione statica del singolo incrocio

La funzione accetta in input *previous*, che è un array di quattro elementi (uno per ogni braccio), nel quale l'unico elemento posto ad uno è quello inerente al braccio che ha ricevuto il verde quando lo script è stato chiamato l'ultima volta. Tutti gli altri valori sono pari a zero. È dunque chiaro che in realtà il vettore in questione si comporta come un canale di feedback, in cui ogni elemento è un flag, e serve per gestire la turnazione.

Come è facile notare, infatti, all'interno dello script viene inizializzato un nuovo array, *current*, sempre di quattro elementi. Si valuta poi quale è stato l'ultimo braccio a cui si è concesso il verde, mediante un *ciclo for*, e si fa scattare il semaforo successivo.

Alla successiva chiamata l'attuale array *current* diventerà a tutti gli effetti il relativo *previous*.

È anche da notare, che oltre alle otto uscite (una per ogni corsia), che in realtà in questo caso statico sono sempre accoppiate (ecco il perché di un vettore *previous* di soli quattro elementi), la funzione manda in output anche il *delay*, ovvero l'intervallo di tempo nel quale le macchine delle strade a cui è stato concesso il verde possono fluire liberamente. Tale *delay* viene poi inviato direttamente al timer, che provvederà a richiamare la funzione alla sua scadenza.

Ogni uscita è poi collegata all'Entity Gate della rispettiva corsia, che, ricevendo 0 si chiude (o resta chiuso), ricevendo 1 permette invece ai veicoli di occupare l'incrocio.

1.5 Analisi dei risultati

Si analizzano dunque qui i risultati di questa gestione, utili poi per un confronto con l'algoritmo di ottimizzazione oggetto di questa tesi. In particolare, i grafici che seguono sono ottenuti con una simulazione in cui i parametri μ per ogni corsia sono i seguenti. Tali grafici, inoltre, si riferiscono alla prima corsia (corsia di sinistra del braccio ovest), dunque al primo parametro di questo vettore.

[33.7997 32.4750 23.2955 33.1698 29.9311 24.1165 31.6948 28.4784]

Come si può notare dalla *figura 1.17*, riferita al numero di auto in coda in funzione dell'ora del giorno considerata, si ha una sensibile variazione fra il numero di auto in coda durante le ore notturne e quello durante le ore diurne, come era lecito aspettarsi visti i tassi di generazione variabili. È interessante notare come il più alto numero di auto in coda sia 8, con una situazione tende ad assestarsi ad un valore leggermente inferiore. È anche importante considerare che, nonostante i tempi di inter-arrivo notturni siano estremamente elevati, come specificato in precedenza, comunque durante la notte tendano ad accodarsi alcune macchine, fino a un massimo di 5.

Questa è una delle spiegazioni per cui nelle grandi città nelle ore meno affollate molti semafori vengono spenti. Come sarà chiaro più avanti, tuttavia, con un algoritmo di gestione dinamica dell'incrocio, il problema dell'accodamento viene risolto, seppur mantenendo acceso l'apparato semaforico, contribuendo alla sicurezza dei guidatori anche a tarda sera.

La *figura 1.18* rappresenta invece il tempo medio di attesa delle auto nella corsia considerata, sempre in funzione dell'ora del giorno. Sebbene il tempo concesso a queste auto per defluire sia alquanto importante (ben 25 secondi), si nota come l'attesa tocchi un picco di quasi 95 secondi, in corrispondenza dell'ora di punta, per poi decrescere, seppur di poco. È altresì fondamentale considerare che, proprio per la mancanza di distinzione nel tempo concesso alla luce verde per ciascuna corsia, anche di notte, con un bassissimo affollamento, si assiste ad un'attesa media di più di 35 secondi.

Appare dunque chiaro come siano ampi i margini di miglioramento, sia per quanto riguarda il numero massimo di auto in coda, che, soprattutto, per quanto concerne i tempi medi di attesa, e nel secondo capitolo di questa trattazione saranno analizzati vari metodi per ottimizzare questi due parametri.

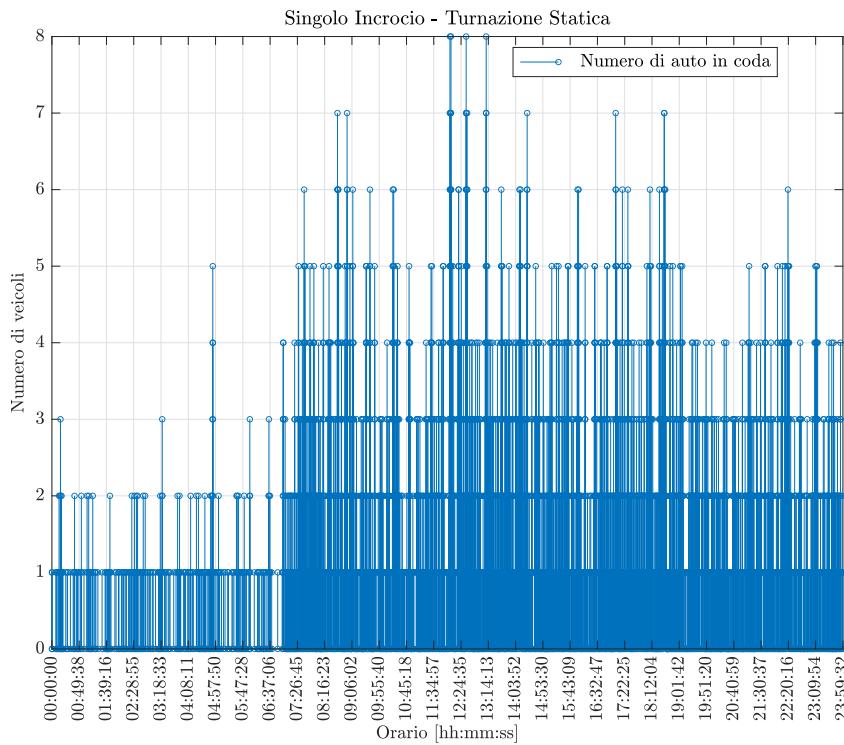


Figura 1.17: Numero di auto in coda per una singola corsia

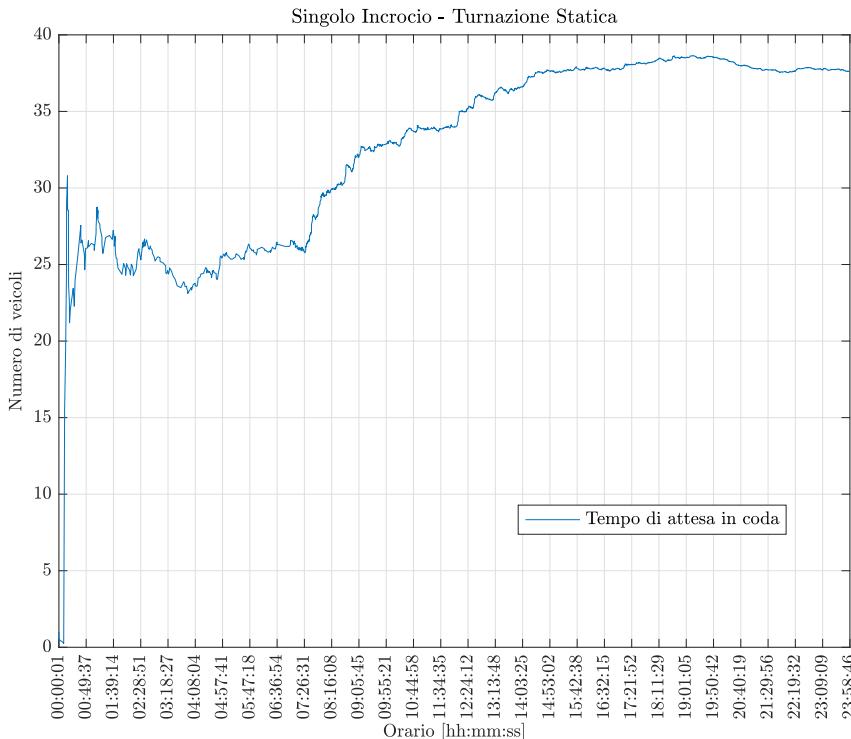


Figura 1.18: Tempi di attesa medi per una singola corsia in funzione dell'orario considerato

1.6 Tabella riassuntiva dei risultati con valori variabili del parametro μ

Nella tabella presente alla seguente pagina sono riportate più esecuzioni dell'algoritmo, sia a fronte dello stesso intervallo relativo al parametro μ , sia per intervalli diversi, per riportare come variano il tempo medio di attesa, il numero di veicoli che attraversano l'incrocio, il numero massimo di veicoli in coda per ogni esecuzione. Tutti questi dati sono inerenti alla singola corsia analizzata, che cambia di esecuzione in esecuzione, come è facile comprendere.

Nello specifico si può notare come nella prima simulazione, con tassi di generazione elevati (mediamente all'ora di pranzo il tempo di arrivo fra una macchina e l'altra è minore di 10 secondi) si accodi un numero di macchine esagerato nella corsia presa in considerazione. La simulazione è stata infatti portata al limite in questo modo, ed è ovvio che un congestionamento di questo genere è alquanto raro, sebbene non inverosimile; l'obiettivo è stato quello di rendere il lettore consapevole del legame fra il parametro μ e tutti gli altri valori relativi alla simulazione stessa. Nel capitolo successivo si confronteranno vari scenari, e sarà più immediato capire come, a parità di μ , qualsiasi sia il suo valore, un algoritmo di gestione dinamica è sicuramente più performante. Risulta anche importante considerare che, a fronte di un sensibile calo del numero di automobili che attraversano l'incrocio, se è vero che cala anche il numero massimo di auto in coda, i tempi di attesa medi restano comunque molto elevati. Sempre al di sopra dei 35 secondi, assolutamente spropositati rispetto a quelli ottenuti con l'algoritmo che verrà proposto nel capitolo successivo.

Prima simulazione					
Intervallo		Stima tempo interarrivo con minimo μ		Stima tempo interarrivo con massimo μ	
[23, 28]		15.94 secondi		19.41 secondi	
<i>Corsia: Ovest - Dritto/Destra</i>					
Esecuzione	μ	Tempo di arrivo medio	Numero di auto totale	Max numero auto in coda	Tempo di attesa medio
1	23.4080	16.25s	3143	39	112.61s
2	24.8866	17.25s	2888	23	67.03s
3	26.2806	18.22s	2809	21	60.08s

Seconda simulazione					
Intervallo		Stima tempo interarrivo con minimo μ		Stima tempo interarrivo con massimo μ	
[28, 33]		19.41 secondi		22.87 secondi	
<i>Corsia: Nord - Dritto/Destra</i>					
Esecuzione	μ	Tempo di arrivo medio	Numero di auto totale	Max numero auto in coda	Tempo di attesa medio
1	32.0241	22.20s	2314	12	39.54s
2	30.4146	21.08s	2381	13	43.12s
3	28.5156	19.77s	2553	13	45.11s

Terza simulazione					
Intervallo		Stima tempo interarrivo con minimo μ		Stima tempo interarrivo con massimo μ	
[33, 38]		22.87 secondi		26.34 secondi	
<i>Corsia: Sud - Sinistra</i>					
Esecuzione	μ	Tempo di arrivo medio	Numero di auto totale	Max numero auto in coda	Tempo di attesa medio
1	37.3261	25.87s	1960	9	35.55s
2	33.6499	23.32s	2206	9	38.69s
3	36.4679	25.28s	1923	7	35.61s

Tabella 1.2: Esecuzioni dell'algoritmo di gestione statica al variare di μ

Capitolo 2

Algoritmo di gestione dinamica applicato ad un singolo incrocio

In questo capitolo si vuole analizzare il comportamento dello stesso incrocio visto in precedenza con un algoritmo di gestione ottimizzata, che conti il numero di auto in ciascuna corsia e in base a questo e ad altri parametri conceda il verde per un numero di secondi variabile. È da precisare che l'algoritmo utilizzato si basa su quello proposto da *Maram Bani Younes e Azzedine Boukerche*[2], tuttavia sono state effettuate alcune importanti modifiche, che verranno analizzate nel corso della trattazione.

Si è previsto, oltre che un raffronto fra l'algoritmo di gestione statica e quello sviluppato in autonomia, modificando quanto proposto dai due ingegneri, anche un meccanismo per valutare l'efficacia di queste modifiche, confrontando il nuovo codice con una implementazione di quello presente nella pubblicazione citata. Tutti i confronti, con i relativi grafici e le relative conclusioni, verranno presentati alla fine di questo capitolo, dopo aver descritto nel dettaglio tutti gli aspetti relativi alle modifiche apportate al modello e ai meccanismi implementativi adottati.

Per quanto riguarda lo pseudocodice progettato dai due ingegneri, è disponibile alla seguente pagina.

```

1 while d_i di una qualsiasi corsia > 0 do
2 {
3
4     sia j la corsia con il maggior numero di macchine;
5     siano i_1 e i_2 le due corsie complementari;
6         if (d_i_1 > d_i_2)
7         {
8             schedula j , i_1 ;
9             d_i_1 = 0;
10            t_i_1 = 0;
11        }
12
13        else
14        {
15            schedula j , i_2 ;
16            d_i_2 = 0;
17            t_i_2 = 0;
18        }
19        d_j = 0;
20        t_j = 0;
21
22 }
```

Codice 2.1: Pseudocodice di gestione dinamica di un incrocio

2.1 Breve spiegazione dell'algoritmo originale

La proposta di *Maram Bani Younes e Azzedine Boukerche* si basa sulla definizione di alcuni parametri, il primo dei quali è la così detta *Ready Area*. Tale area è sostanzialmente il range entro il quale le macchine si possono considerare in coda per l'incrocio: si individua, per ogni semaforo della giunzione, una distanza massima, di solito corrispondente alla portata del dispositivo di rilevazione in termini di veicoli, oltre la quale le vetture non vengono più conteggiate.

Il dispositivo in questione può essere rappresentato da una semplice telecamera che, mediante meccanismi tipici della Computer Vision, utilizzando API quali Google Vision[17] o OpenCV[18], riesca ad individuare la variabile principale su cui si basa l'algoritmo: il numero di auto in coda per ciascuna corsia. L'implementazione hardware di questo sistema, tuttavia, non è oggetto di questa tesi e dunque non ci si dilungerà oltre nella sua spiegazione.

Il tempo concesso alle corsie prescelte (durata della luce verde) è poi dinamico, e viene calcolato sulla base della seguente formula:

$$T = \theta + \frac{F_d}{S_{tf}} \quad (2.1)$$

Il parametro θ è una costante, e rappresenta il tempo che mediamente è necessario alla prima macchina per partire, mentre F_d è la distanza del veicolo più lontano dal semaforo (sempre interno alla *Ready Area*). In ultimo, S_{tf} è la velocità media del flusso di auto nell'intersezione. Come è ovvio notare, sia θ che S_{tf} sono ottenuti mediante delle stime, non essendo misurabili prima della effettiva esecuzione dell'algoritmo.

Una proposta alternativa, certamente più efficiente ma anche meno facile da implementare, è quella di permettere alle automobili di comunicare fra loro e con il semaforo dati quali la F_d (distanza), nonché la loro velocità media. Certamente in questo modo si renderebbe l'esecuzione dello script più veritiera ed affidabile, ed in futuro, grazie agli sforzi delle case automobilistiche, probabilmente tutto ciò sarà possibile.

Altri parametri da considerare sono d_i che rappresenta il numero di automobili in coda nella corsia i-esima e t_i , tempo richiesto a tutti i veicoli della corsia i-esima (all'interno della *Ready Area*) per lasciare l'intersezione.

Il funzionamento dell'algoritmo è dunque il seguente: si individua la corsia più affollata fra quella che presentano almeno un'automobile, e contestualmente si rintracciano le due corsie compatibili con essa, così come spiegato nel *Capitolo 1* e come si può notare nella *figura 1.1*.

A questo punto, fra queste due candidate si sceglie ancora la più affollata; in questo modo non solo si va a concedere il verde alla strada con più automobili, ma si cerca anche di ottimizzare la scelta della sua complementare, che non resta statica (come invece è nell'algoritmo utilizzato nel capitolo precedente), ma diventa dinamica ed intelligente. Successivamente l'algoritmo prevede di calcolare il tempo T , concedendolo alla luce verde delle due corsie, e di azzerare le variabili d_i, d_j e t_i, t_j relative, in quanto si prevede che dopo T secondi le corsie in questione presentino un numero di veicoli pari a zero.

2.2 Le modifiche apportate

Sono molteplici le modifiche apportate all'algoritmo proposto, alcune delle quali a puro scopo implementativo, altre come migliorie.

In primo luogo, si è preferito non considerare una *Ready Area* per non dover dare alle code di *Simulink* un massimo numero di entità (veicoli) da contenere, questo per effettuare più simulazioni variando i tassi di generazione, verificando il comportamento del modello anche in condizioni esageratamente negative, con code anche molto affollate (più di 50 automobili).

Il concetto della *Ready Area*, tuttavia, non è stato abbandonato del tutto, ma è stato piuttosto migliorato: si è deciso di concedere un tempo minimo ed un tempo massimo alla durata del verde per ciascuna corsia. Il tempo minimo serve perché, anche con una sola macchina in coda, si è voluto stabilizzare la situazione, che altrimenti si sarebbe tradotta in un passaggio troppo repentino dal verde al rosso. Per quanto concerne il tempo massimo, questo è diretta espressione della *Ready Area*: invece che considerare un massimo numero di veicoli, si considera che, indipendentemente da questo numero, il verde non può durare più di un certo valore, questo per evitare tempi di attesa troppo lunghi nelle corsie meno affollate, ma comunque in cui sono presenti veicoli.

Il tempo concesso alla luce verde, poi, viene calcolato tenendo conto della distribuzione uniforme che governa gli *Entity Server*, precedentemente descritta, assegnando ad ogni macchina un valore pari al valore medio della distribuzione (in questa simulazione pari a $5s$), ed ottenendo il tempo complessivo moltiplicando questo valore per il numero di veicoli in coda. A questo punto, se il numero ottenuto è inferiore al tempo minimo, esso si scarta, e si prende in considerazione proprio quel minimo, viceversa se supera il tempo massimo è questo ad essere tenuto in considerazione e ad essere concesso. Pertanto, poiché non è detto che le code a cui viene dato il via libera si svuotino, non si va ad azzerare i contatori relativi, ma semplicemente all'esecuzione successiva della funzione questi sono aggiornati con i nuovi dati rilevati, ottenuti dalle code stesse.

Inoltre, è stata individuata una grande lacuna nell'algoritmo oggetto della pubblicazione citata: l'assenza di un meccanismo di gestione della *starvation*.

Considerando esclusivamente il *codice 2.1*, infatti, si nota che se in una strada continuano a confluire veicoli più velocemente che nelle altre, a questa sarà sempre concesso il verde. Questo comportamento da un lato è positivo perché si va a decongestionare tale corsia, ma è ovviamente improponibile, in quanto potrebbe paradossalmente capitare che un semaforo resti rosso per ore ed ore, e che una persona

sia costretta ad attendere un tempo esagerato prima di passare, perché la sua strada è meno affollata.

È stato scelto dunque di implementare un meccanismo di gestione della *starvation* mediante un algoritmo *LRU* (Least Recently Used) opportunamente modificato. Ad ogni corsia è associato un contatore, ed ogni qual volta ad una strada viene concesso il verde, il suo contatore viene azzerato, mentre quello delle altre si incrementa del numero di secondi per cui è stato concesso il verde alla corsia suddetta. In questo modo, quando un contatore raggiunge un valore limite (liberamente impostabile), la relativa strada ottiene priorità massima, e l'algoritmo viene applicato in maniera identica (calcolo del tempo del verde e selezione della complementare) come se essa fosse quella più affollata. In particolare il tempo massimo di attesa prima che scatti la starvation è stato impostato a 5 minuti.

È anche da notare che sono stati apportati alcuni minimi cambiamenti al modello: le code sono state collegate agli ingressi del blocco contenente la funzione da eseguire, comunicando ad essa il numero di veicoli presenti, ed il canale di *feedback* è stato utilizzato per implementare il meccanismo della gestione della *starvation*, e non per conoscere a quale semaforo è stato concesso il verde nell'esecuzione precedente, informazione ora inutile. Non è stato variato altro, se non, ovviamente, il codice nel blocco *Matlab Function*, quindi non si ravvisa la necessità di approfondire nuovamente la necessità di descrivere nei dettagli nuovamente l'intero modello, il cui schema è comunque presente alla pagina seguente, seguito dal codice che lo governa.

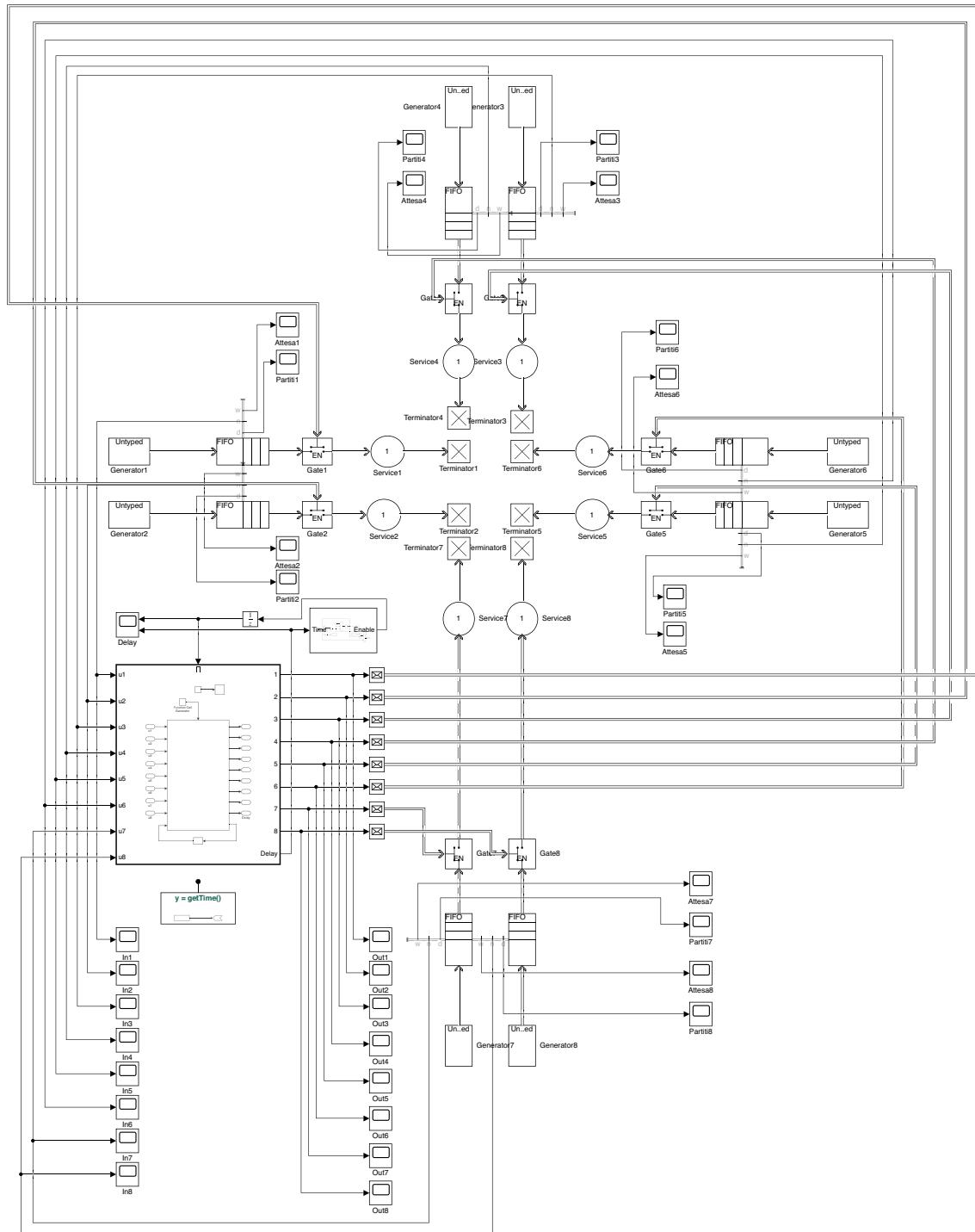


Figura 2.1: Modello di un incrocio a raso a quattro bracci in Simulink e SimEvents, gestione dinamica

```

1 function [ delay , y1 , y2 , y3 , y4 , y5 , y6 , y7 , y8 , feedback ] =
2     fcn(u1 , u2 , u3 , u4 , u5 , u6 , u7 , u8 , history)
3
4 % VARIABILI DI USCITA
5 % Prima attesa , prima dell ' esecuzione dell ' algoritmo la
6 % prima volta
7 delay = 1;
8 y1 = 0;
9 y2 = 0;
10 y3 = 0;
11 y4 = 0;
12 y5 = 0;
13 y6 = 0;
14 y7 = 0;
15 y8 = 0;
16 feedback = zeros([4 2]);
17
18 % Numero di veicoli per ogni direzione
19 M = ([ u1 u2 ; u3 u4 ; u5 u6 ; u7 u8 ; ]) ;
20
21 % Tempo medio per fluire per ogni macchina
22 Tv = 5;
23
24 % Massimo tempo del verde
25 MaxGreen = 60;
26
27 % Minimo tempo del verde
28 MinGreen = 10;
29
30 % Controllo sulla starvation (LRU) . Tempo di attesa massimo :
31 % 5 minuti
32 starvation = 300;
33
34 % Cerco il massimo
35 [R, C] = size(M);
36 iMax = 1;

```

```

35 jMax = 1;
36 vMax = M(1, 1);
37
38 %Usato per la starvation, interrompe la ricerca
39 flag = 0;
40
41 for i = 1 : 1 : R
42     for j = 1 : 1 : C
43
44         if M(i, j) > 0 && history(i, j) > starvation
45             vMax = M(i, j);
46             iMax = i;
47             jMax = j;
48             flag = 1;
49             break;
50     end
51
52     if M(i, j) > vMax
53         vMax = M(i, j);
54         iMax = i;
55         jMax = j;
56     end
57
58 end
59
60 if flag == 1
61     break;
62 end
63 end
64
65
66 % Cerco i possibili candidati
67 [c1_i, c1_j, c2_i, c2_j] = getCandidates(iMax, jMax);
68
69 %Scelgo tra i candidati
70 if M(c1_i, c1_j) > M(c2_i, c2_j)
71     iCan = c1_i;

```

```

72     jCan = c1_j ;
73 else
74     iCan = c2_i ;
75     jCan = c2_j ;
76 end
77
78 delay = vMax*Tv;
79
80 if vMax*Tv > MaxGreen
81     delay = MaxGreen;
82 elseif vMax*Tv < MinGreen
83     delay = MinGreen ;
84 end
85
86 %Risultato
87 U = ([0 0; 0 0; 0 0]);
88
89 U(iMax , jMax) = 1;
90 U(iCan , jCan) = 1;
91
92 history( : ) = history( : ) + delay ;
93
94 history(iMax , jMax) = 0;
95 history(iCan , jCan) = 0;
96
97 %Output
98 y1 = U(1 , 1); y2 = U(1 , 2); y3 = U(2 , 1); y4 = U(2 , 2); y5 =
    U(3 , 1); y6 = U(3 , 2); y7 = U(4 , 1); y8 = U(4 , 2);
    feedback(:) = history(:);
99 end

```

Codice 2.2: Implementazione dell'algoritmo di gestione dinamica di un singolo incrocio

2.3 Spiegazione del codice implementato

In primo luogo vengono inizializzate le variabili in uscita della funzione, corrispondenti al valore di trigger degli otto entity gate rappresentanti le varie corsie. Di default questi valori sono posti a 0 (semaforo rosso). In input si accetta il numero di veicoli presenti in ogni coda nell'istante in cui la funzione viene richiamata, nonché il vettore *history*, usato per gestire la starvation, il cui significato sarà chiaro più avanti.

I due *cicli for* annidati servono ad individuare la strada con il numero di veicoli più elevato, purché non si verifichi il caso di *starvation*. Tale strada (individuata da due indici, i e j all'interno della matrice M , matrice che organizza gli otto input indipendenti), viene poi data in pasto alla funzione *getCandidates*, il cui codice è alla pagina seguente e che restituisce gli indici relativi alla matrice M che individuano le due strade che possono essere scelte assieme alla principale. A questo punto fra le due corsie complementari candidate viene scelta, come già detto, quella più affollata. Viene calcolato il tempo da concedere al verde in funzione del numero di veicoli della strada scelta come principale, e vengono effettuati i controlli con il tempo minimo ed il tempo massimo, precedentemente dichiarati come costanti.

Successivamente, ogni elemento del vettore *History*, il quale contiene i contatori inerenti all'algoritmo *LRU* per ogni corsia, viene incrementato del tempo stabilito, azzerando poi gli elementi corrispondenti alle corsie a cui si sta concedendo il verde. In output viene inviato *1* agli *Entity Gate* che devono aprirsi, corrispondenti alle strade individuate, e *0* a tutti gli altri.

Oltre a queste informazioni rappresenta un output anche il *delay*, ovvero proprio il tempo concesso, che viene inviato al timer descritto nel *capitolo 1*, che richiamerà la funzione allo scadere dello stesso.

```

1   function [ i1 , j1 , i2 , j2 ] = getCandidates( i , j )
2     % Direzioni accoppiate (8x4)
3     % Per ciascuna direzione scelta , ci sono 2 possibili
4       direzioni
5     % complementari
6     %  $D = \{[1 \ 2; \ 3 \ 1] \ [1 \ 1; \ 3 \ 2]; \ [4 \ 1; \ 2 \ 2] \ [2 \ 1; \ 4 \ 2]; \ [3 \ 2; \ 1 \ 1] \ [3 \ 1; \ 1 \ 2]; \ [4 \ 2; \ 2 \ 1] \ [4 \ 1; \ 2 \ 2]\};$ 
7
8     i1 = 0;
9     i2 = 0;
10    j1 = 0;
11    j2 = 0;
12
13    if i==1 && j ==1
14      i1 = 1;
15      j1 = 2;
16      i2 = 3;
17      j2 = 1;
18    elseif i==1 && j==2
19      i1 = 1;
20      j1 = 1;
21      i2 = 3;
22      j2 = 2;
23    elseif i==2 && j==1
24      i1 = 4;
25      j1 = 1;
26      i2 = 2;
27      j2 = 2;
28    elseif i==2 && j==2
29      i1 = 2;
30      j1 = 1;
31      i2 = 4;
32      j2 = 2;
33    elseif i==3 && j==1
34      i1 = 3;
35      j1 = 2;
36      i2 = 1;

```

```

36    j2 = 1;
37    elseif i==3 && j==2
38        i1 = 3;
39        j1 = 1;
40        i2 = 1;
41        j2 = 2;
42    elseif i==4 && j==1
43        i1 = 4;
44        j1 = 2;
45        i2 = 2;
46        j2 = 1;
47    elseif i==4 && j==2
48        i1 = 4;
49        j1 = 1;
50        i2 = 2;
51        j2 = 2;
52    end
53
54
55 end

```

Codice 2.3: Scelta delle due strade complementari a quella selezionata come principale

È da sottolineare che questo codice funziona fintanto che gli ingressi non vengono variati: se si dovesse collegare una strada ad un ingresso diverso da quello previsto ed utilizzato nella simulazione chiaramente l'intero algoritmo non sarebbe più affidabile. L'autore di questa tesi è consapevole che esistono metodi più eleganti ed ottimizzati rispetto a questa sequenza di *if* ed *elseif* per ottenere il medesimo scopo, tuttavia si è reso necessario procedere in questo modo per lo scarso supporto del linguaggio *Matlab* a tali meccanismi, come, per esempio, gli *switch - case*. In questo modo, inoltre, il codice risulta particolarmente leggibile e facile da comprendere.

Le differenze con un normale algoritmo di gestione di una giunzione, come quello proposto nel *capitolo 1* sono quindi evidenti: in primo luogo non si segue una turnazione statica per quanto riguarda la concessione del verde, seguendo invece un meccanismo di priorità: ci si chiede infatti quale sia la strada che necessita del verde in quell'istante, più delle altre, e glielo si concede. Inoltre anche il tempo per cui questo verde è concesso è variabile: si ipotizzi per esempio di analizzare l'incrocio durante le ore notturne, non avrebbe senso concedere il verde per un numero di

secondi elevato quando in coda è presente una sola macchina, si trattrebbe di una perdita di tempo, tradotta in un maggior consumo di carburante ed un maggiore stress per gli automobilisti delle altre corsie, costretti ad attendere perché viene concesso il verde ad una strada praticamente vuota. Viceversa è ovvio che durante il giorno, per una strada affollata, la durata della luce verde deve essere sufficiente da garantire che un buon numero delle macchine in coda defluisca.

2.4 Il modello per confrontare le diverse gestioni del singolo incrocio

I benefici esposti, ovviamente, devono trovare riscontro in un confronto imparziale ed assolutamente valido fra i modelli presentati, affinché a partire dai medesimi dati in input (le automobili generate), e con gli stessi parametri (tasso di generazione, tempo di servizio interno agli *Entity Server*) essi gestiscano l'incrocio secondo la propria logica. Si è scelto, pertanto, di creare in un nuovo workspace il modello identificato dalla *figura 2.2*. Si può notare come al suo interno siano presenti tre blocchi principali, ognuno dei quali è rappresentativo di un algoritmo di gestione. Guardando infatti ai tre blocchi presenti, si comprende subito come questi siano in realtà esattamente i modelli precedentemente descritti. Il primo, quello più in alto a sinistra, è quello presentato nel *capitolo 1*, relativo ad una gestione statica (*figura 1.5*). Quello in basso a sinistra invece si riferisce al modello di controllo dinamico, di cui si è parlato in questo capitolo (*figura 2.1*), mentre quello in basso a destra è perfettamente identico a quest'ultimo, eccezion fatta per la gestione della starvation, che si è eliminata. Ciò è stato fatto per capire come tale gestione, che comunque resta assolutamente necessaria, interferisca con il normale funzionamento dell'algoritmo stesso.

Nello specifico, in questo terzo modello, si è provveduto semplicemente a commentare la seguente porzione di codice, lasciando tutto il resto invariato.

```
1  if M(i , j ) > 0 && history(i , j ) > starvation
2      vMax = M(i , j );
3      iMax = i ;
4      jMax = j ;
5      flag = 1;
6      break;
7  end
8      ...
9      ...
10     ...
11 if flag == 1
12     break;
13 end
```

Codice 2.4: Porzione di codice relativa alla gestione della starvation

Gli *Entity Generator*, poi, sono stati collegati a dei blocchi di tipo **Entity Replicator[19]**, in modo tale che appena una entità viene generata in una corsia specifica, questa venga subito inviata ai tre modelli contemporaneamente. In questo modo i tre incroci sono esattamente identici sia per quanto riguarda il numero di entità che devono gestire che per quanto concerne quando queste si presentano, tutto questo per garantire l'assoluta imparzialità delle condizioni, precedentemente citata.

Dunque il workflow è il seguente: quando una entità viene generata, questa viene inviata a tutti e tre gli incroci, in corrispondenza della stessa corsia e nello stesso istante di tempo. La simulazione parte allo stesso istante per tutti e tutti sono configurati secondo gli stessi parametri. Al termine della simulazione tutti gli incroci producono tre grafici per ogni corsia, relativi al tempo di attesa medio, al numero di macchine totali che hanno attraversato tale strada ed al numero medio di macchine in coda in funzione dell'ora del giorno. Confrontando i grafici di una stessa strada, e facendo questo per ogni strada presente, si ottiene una chiara idea di quale dei tre algoritmi stia funzionando meglio.

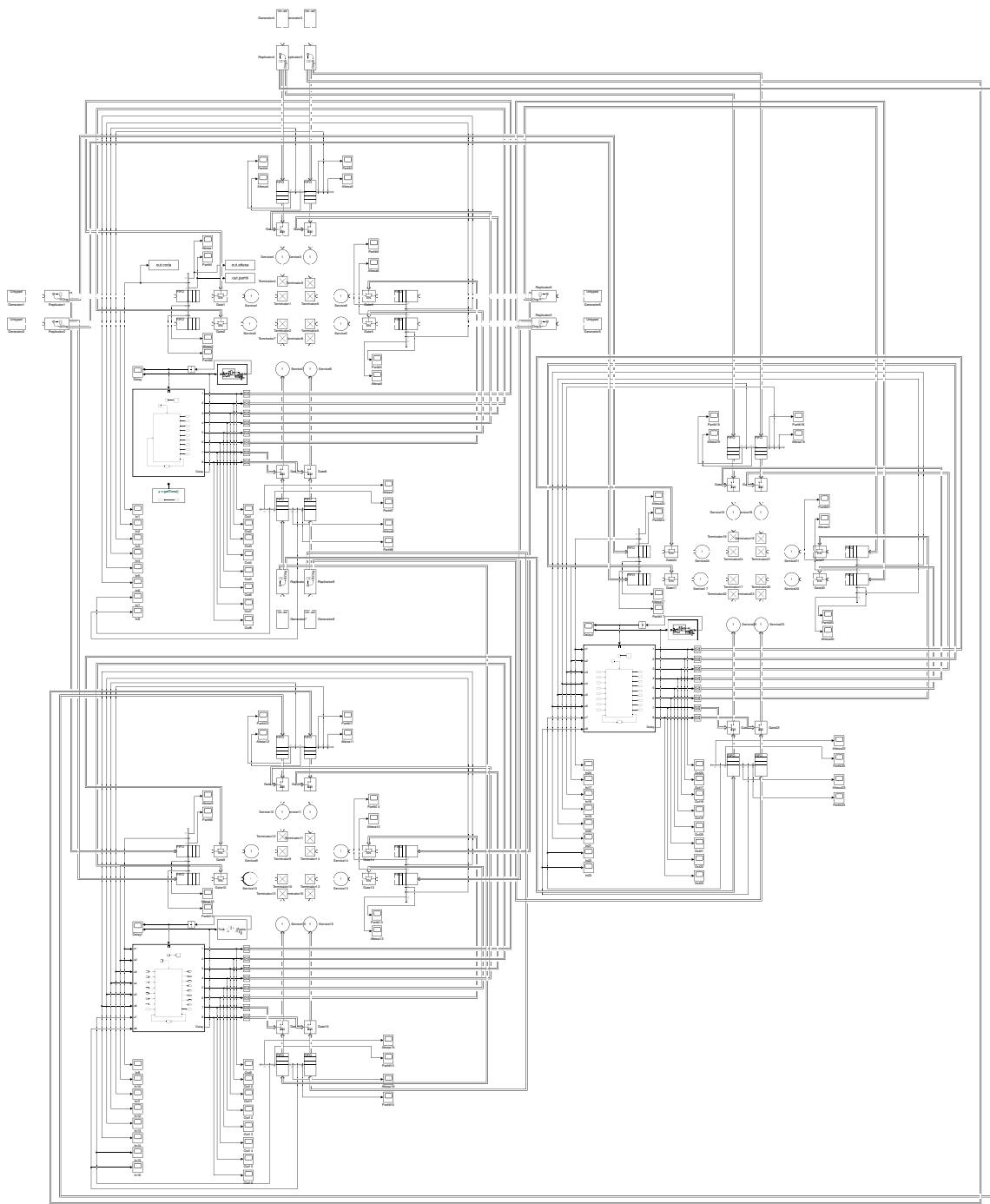


Figura 2.2: Modello per comparare gli algoritmi utilizzati per la gestione di un singolo incrocio

2.5 I risultati ottenuti

Per le seguenti simulazioni il primo scenario, ovvero il seguente, presenta dei tassi di generazione randomici, esattamente come nel *capitolo 1*, con il parametro μ scelto casualmente, secondo una distribuzione uniforme, fra 23 e 35. Questo per ottenere un numero di automobili sì variabile, ma che sia confrontabile fra le varie strade, con una configurazione dunque che può dirsi bilanciata. Il codice interno agli *Entity Generator* non è dunque stato variato rispetto a quanto presentato nel *codice 1.2*, così come la definizione dei parametri globali (*codice 1.1*). In una configurazione di questo genere sono stati ottenuti i risultati riportati nelle seguenti tabelle, a fronte di questi valori di μ generati.

Prima simulazione: prime quattro corsie						
Intervallo	Stima tempo interarrivo con minimo μ			Stima tempo interarrivo con massimo μ		
[23, 35]	15.94s			24.26		
<i>Corsia Ovest — Sinistra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	24.1705		16.75s		2940	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	25	59.44s	11	19.18s	13	19.07s
<i>Corsia Ovest — Dritto/Destra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	26.3420		18.26s		2809	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	18	52.61s	8	17.27s	10	17.34s
<i>Corsia Nord — Sinistra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	29.5626		20.49s		2384	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	10	38.86s	9	24.02s	10	22.85s
<i>Corsia Nord — Dritto/Destra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	34.4901		23.91s		2049	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	11	37.12s	8	21.31s	7	21.43s

Tabella 2.1: Tabella di comparazione fra algoritmi di gestione del singolo incrocio - prime quattro corsie

Prima simulazione: ultime quattro corsie						
Intervallo	Stima tempo interarrivo con minimo μ			Stima tempo interarrivo con massimo μ		
[23, 35]	15.94s			24.26		
<i>Corsia Est — Sinistra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	34.5787		23.97s		2152	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	9	36.93s	7	18.76s	8	19.08s
<i>Corsia Est — Dritto/Destra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	24.8914		17.25s		2928	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	32	65.24s	11	22.33s	11	22.48s
<i>Corsia Sud — Sinistra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	34.6471		20.02s		1996	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	11	37.92s	8	27.96s	10	27.16s
<i>Corsia Sud — Dritto/Destra</i>	μ		Tempo di arrivo medio		Numero di auto totale	
	34.4860		23.90s		2175	
	Gestione statica		Gestione dinamica senza starvation		Gestione dinamica con starvation	
	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio	Max numero auto in coda	Tempo di attesa medio
	10	36.63s	11	29.44s	8	27.69s

Tabella 2.2: Tabella di comparazione fra algoritmi di gestione del singolo incrocio - ultime quattro corsie

Sono riportati, inoltre, alcuni grafici indicativi, relativi ad una singola corsia (sempre la stessa, quella del braccio ovest relativa alla svolta a sinistra), per ciascuno dei tre modelli, inerenti al tempo medio d'attesa ed al numero di macchine in coda in funzione dell'ora del giorno.

TABELLA CON MU FISSI PER CONGESTIONARE

Bibliografia

- [1] *Intersezione a raso a quattro bracci.* URL: https://it.wikipedia.org/wiki/Intersezione_a_raso.
- [2] Azzedine Boukerche Maram Bani Younes. «Intelligent Traffic Light Controlling Algorithms Using Vehicular Networks». In: *IEEE Transactions on Vehicular Technology* (2016), pp. 5887–5889.
- [3] Daniel L. Gerlough. *The Probability Theory Applied to Distribution of Vehicles on Two-Lane Highways*. 1995.
- [4] *Entity Generator.* URL: <https://www.mathworks.com/help/simevents/ref/entitygenerator.html>.
- [5] *Entity Queue.* URL: <https://www.mathworks.com/help/simevents/ref/queue.html>.
- [6] *Entity Gate.* URL: <https://www.mathworks.com/help/simevents/ref/entitygate.html>.
- [7] *Entity Server.* URL: <https://www.mathworks.com/help/simevents/ref/entityserver.html>.
- [8] *Entity Terminator.* URL: <https://www.mathworks.com/help/simevents/ref/entityterminator.html>.
- [9] *Enabled Subsystem.* URL: <https://www.mathworks.com/help/simulink/slref/enabledsubsystem.html>.
- [10] *Function-Call Subsystem.* URL: <https://www.mathworks.com/help/simulink/slref/functioncallsubsystem.html>.
- [11] *Message Send.* URL: <https://www.mathworks.com/help/simulink/slref/send.html>.
- [12] *Function-Call Feedback Latch.* URL: <https://www.mathworks.com/help/simulink/slref/functioncallfeedbacklatch.html>.
- [13] *Subsystem.* URL: <https://www.mathworks.com/help/simulink/slref/subsystem.html>.

- [14] *Unit Delay*. URL: <https://www.mathworks.com/help/simulink/slref/unitdelay.html>.
- [15] *Simulink Function*. URL: <https://www.mathworks.com/help/simulink/simulink-functions-in-simulink-models.html>.
- [16] *Digital Clock*. URL: <https://www.mathworks.com/help/simulink/slref/digitalclock.html>.
- [17] *API Google Vision*. URL: <https://cloud.google.com/vision?hl=it>.
- [18] *API OpenCV*. URL: <https://opencv.org>.
- [19] *Entity Replicator*. URL: <https://it.mathworks.com/help/simevents/ref/entityreplicator.html>.