SAPIENZA
UNIVERSITÀ DI ROMA

JTrafficControl

Tesina per il corso di "Metodi Formali nell'Ingegneria del Software"

Docente: Toni Mancini

A.A. 2006-2007

Autori: Raffaele Giuliano, Marco Piva, Fabio Terella, Emanuele Tracanna

Abstract

La seguente relazione mostra come sia possibile coniugare metodologie e tecniche del mondo dei metodi formali, con un caso concreto e pratico. Il caso concreto è quello dello scheduling di un incrocio semaforico, con tutti i problemi derivanti dalla presenza di un numero arbitrario di strade e di corsie, nonché la necessità di dover esplicitare vincoli temporali che regolano la durata del rosso o del verde di un qualunque semaforo. Le metodologie e le tecniche del mondo dei metodi formali sono quelle proprie delle logiche temporali e di NuSMV, un potente Model Checker, in grado di manipolare queste logiche e di fornire una struttura temporale che soddisfi tutte le formule dichiarate.

Il lavoro mostra quindi come sia possibile creare delle formule in logica temporale in grado di descrivere un incrocio semaforico, e di verificarle tramite NuSMV, con il fine di ottenere uno schedule rispettoso di tutti i vincoli imposti.

Introduzione

La tesina JTrafficControl nasce da una precisa esigenza: coniugare i contenuti del corso di Metodi Formali per l'Ingegneria del Software con un esempio pratico e funzionante. Già dal nome si intuisce che il dominio applicativo di questo lavoro è quello del traffico veicolare, esattamente del controllo e scheduling di un incrocio semaforico, con tutte le possibili varianti ammesse.

Un ipotetico scenario illustrativo potrebbe essere quello di un operatore che deve trovare la corretta allocazione dei semafori per far funzionare un incrocio a più vie, facendo rispettare al contempo alcuni vincoli funzionali, come per esempio la durata minima del verde (o la massima del rosso), la durata minima degli attraversamenti pedonali etc etc… Il sistema in effetti permette di inserire un numero arbitrario di strade confluenti nell'incrocio e, per ognuna di queste strade, può specificare il numero di corsie e la tipologia di traffico. Una volta specificata la natura dell'incrocio, l'operatore può selezionare i vincoli più idonei per la caratterizzazione stessa dell'incrocio e lanciare l'esecuzione del programma, che cercherà una soluzione soddisfacente per il modello creato. L'operatore potrà quindi verificare, attraverso la lettura dell'output proposto, qual è lo scheduling trovato, con l'evidenziazione istante per istante dei flussi veicolari o pedonali abilitati e fermati.

La ricerca della soluzione al problema di scheduling dell'incrocio semaforico è il *gancio* operativo con il corso di metodi formali perché è grazie ad uno strumento del corso che siamo riusciti a risolvere il problema, e di conseguenza a presentare questo lavoro. La tesina effettivamente si basa sull'utilizzo di logiche temporali e di un *Symbolic Model Checker* come NuSMV che, se utilizzato propriamente, oltre a verificare che una formula in logica temporale sia o meno una tautologia, riesce anche a trovare un controesempio pratico che soddisfi il modello[1]. La tesina intuitivamente dato un incrocio, crea il modello usando strutture delle logiche temporali e lo manda in pasto a NuSMV, il quale attraverso una specifica computazione ricerca un controesempio soddisfacente il modello, ovvero ritorna un possibile scheduling dell'incrocio.

La struttura di questo documento riflette tutti i passi effettuati per l'approccio al problema e la sua risoluzione: dapprima prenderemo in esame la rappresentazione del problema e del modello utilizzato, con annessa traduzione in un file SMV, successivamente analizzeremo la struttura del programma ed il suo funzionamento, per terminare infine con l'illustrazione di alcuni esempi pratici.

[1] In realtà NuSMV verifica solamente se una formula in logica temporale sia o meno una tautologia però, se la formula viene negata, NuSMV può essere utilizzato per la verifica della non validità della formula negata, e quindi della soddisfacibilità della formula, fornendo un controesempio valido.

Descrizione del problema

L'obiettivo di questo lavoro è quello di creare un sistema che, dato in input un incrocio, trovi uno scheduling per i vari semafori che regoli il traffico rispettando vari requisiti, dettati da un lato dalla topologia dell'incrocio, dall'altro da esplicite specifiche da parte dell'utente. La prima cosa che abbiamo dovuto fare, quindi, è stata la definizione di cosa l'utente potesse dare in input al programma. Tali input sono:

- La topologia dell'incrocio, con la specifica delle varie strade che lo compongono, delle varie corsie che compongono ciascuna strada, delle tipologie di traffico ammesse nelle varie corsie, degli attraversamenti pedonali eventualmente presenti e della possibilità di effettuare o meno un'inversione ad U sulle varie strade.
- La lista degli eventuali flussi di traffico non consentiti. Ad esempio, l'utente può decidere che in una determinata strada, la svolta a destra è vietata.
- La specifica di vincoli temporali sui vari semafori. Questi possono essere di due tipi, vincoli di durata minima del verde e vincoli di durata massima del rosso, e possono essere espressi rispetto ai singoli flussi oppure rispetto alle varie tipologie di traffico.

Basandosi sugli input dell'utente, la determinazione dello scheduling dovrà tenere conto dei seguenti requisiti:

- Ciascun semaforo che regola un flusso di traffico, deve diventare verde un numero infinito di volte (o, in altre parole, in ogni momento deve essere vero che prima o poi il semaforo diventerà verde).
- Se due flussi semaforici collidono tra loro, allora non deve mai essere possibile che i loro semafori siano contemporaneamente verdi.
- Ciascun semaforo deve rispettare gli eventuali vincoli di durata minima del verde, imposti dall'utente.
- Ciascun semaforo deve rispettare gli eventuali vincoli di durata massima del rosso, imposti dall'utente.

Il modello

Al fine di risolvere il problema dello scheduling, è stato necessario creare un modello che permettesse di esprimere i vari elementi dell'incrocio e allo stesso tempo rappresentare efficacemente i vincoli determinati dai requisiti precedentemente elencati.

Per spiegare il modello è utile far riferimento ad un esempio, rappresentato nella seguente figura.

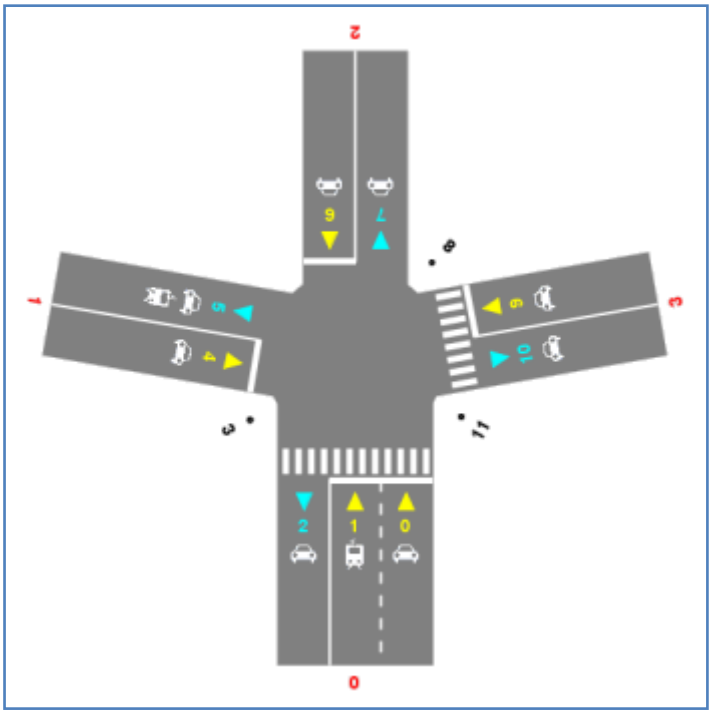


Figura 1 - Esempio di incrocio

Un incrocio è definito come un insieme di strade. Una strada è definita come un insieme di corsie, che può avere o meno un attraversamento pedonale e può permettere o meno l'inversione di marcia. Una corsia può essere entrante o uscente, a seconda che entri o rispettivamente esca dall'incrocio. Ogni corsia permette il transito di uno o più tipi di traffico veicolare. In particolare, nel nostro modello, una corsia entrante può permettere il transito di un solo tipo di traffico, con il vincolo che una strada può avere al più una corsia entrante per ogni tipo di traffico. Una corsia uscente, invece, può permettere il transito di più di un tipo di traffico, con il vincolo che una strada può avere al più una corsia uscente per ogni tipo di traffico. Questi vincoli imposti sulle corsie potrebbero a prima vista sembrare delle limitazioni: in realtà non è così, poiché, non si tratta di corsie "fisiche", ma di corsie "logiche" (ciascuna corsia logica può corrispondere a più corsie fisiche). L'utilizzo di corsie logiche, rende più semplice la modellazione dei vari flussi di traffico, senza per questo diminuire l'insieme degli incroci rappresentabili.

Scelta una strada di riferimento arbitraria, a cui viene assegnato l'indice 0, le strade vengono numerate con indici crescenti procedendo in senso orario in base alla topologia dell'incrocio. Dopodiché, si considerano tre tipologie di punti:

- Sorgenti: consideriamo un punto per ogni corsia entrante nell'incrocio, con il vincolo che una strada può avere al più una corsia entrante per ciascun tipo di traffico (auto o tram) e che ciascuna corsia entrante può ospitare un solo tipo di traffico.
- Destinazioni: consideriamo un punto per ogni corsia uscente dall'incrocio, con il vincolo che una strada può avere al più una corsia uscente per ciascun tipo di traffico (auto o tram) e che ciascuna corsia uscente può ospitare anche più tipi di traffico (cfr. figura: la corsia 5 permette la circolazione sia delle auto che dei tram).
- Punti di attraversamento pedonale: consideriamo un punto aggiuntivo, apposto tra due strade adiacenti ogniqualvolta almeno una delle due strade ha un attraversamento pedonale (cfr. figura: tra la strada 0 e la 1 c'è il punto di attraversamento 3 poiché almeno una delle due strade, e cioè la 1, ha l'attraversamento).

Anche tali punti sono numerati con indici crescenti procedendo in senso orario, a partire dalla prima corsia entrante della strada 0.

Definizione – Insiemi rilevanti

Dati i punti precedentemente individuati, possiamo evidenziare alcuni insiemi che ci serviranno per le definizioni seguenti.

- $T = \{insieme\ di\ tutti\ i\ tipi\ di\ traffico\ veicolare\ (esclusi\ quindi\ i\ pedoni)\}$

Per ogni tipo di traffico (eccetto quello pedonale), $TipoTraffico \in T$, definiamo gli insiemi:

- $S_{TipoTraffico} = \{insieme\ delle\ sorgenti\ che\ consentono\ il\ transito\ di\ TipoTraffico\}$
- $D_{TipoTraffico} = \{insieme\ delle\ destinazioni\ che\ consentono\ il\ transito\ di\ TipoTraffico\}$

Definiamo inoltre gli insiemi:

- $S = \bigcup_{TipoTraffico\ \in T} S_{TipoTraffico} = \{insieme\ di\ tutte\ le\ sorgenti\}$
- $D = \bigcup_{TipoTraffico\ \in T} D_{TipoTraffico} = \{insieme\ di\ tutte\ le\ destinazioni\}$
- $A = \{insieme\ dei\ punti\ di\ attraversamento\}$

Definizione – Flusso

Data la rappresentazione dell'incrocio precedentemente descritta, un flusso di traffico è rappresentato come una coppia di indici (s, d), in cui:

- $s \in S \cup A$ è l'indice del punto di partenza del flusso (sorgente);
- $d \in D \cup A$ è l'indice del punto di arrivo del flusso (destinazione).

Con riferimento alla figura 1, il flusso (0,5) identifica il flusso di traffico che parte dalla corsia di indice 0 ed arriva alla corsia di indice 5.

La definizione di flusso appena fornita è molto generale e consente anche il collegamento di sorgenti e destinazioni di diverso tipo di traffico. Attraverso le definizioni seguenti, la nozione di flusso sarà raffinata, fino ad arrivare al concetto di *flusso ammissibile*, che è quello veramente rilevante ai fini del problema.

Esistono due tipi principali di flusso: i flussi veicolari e i flussi pedonali.

Definizione – Flusso veicolare

Un flusso (s, d) si dice **veicolare** di tipo $TipoTraffico \in T$ se $s \in S_{TipoTraffico}$ e $d \in D_{TipoTraffico}$. Ad esempio un flusso veicolare è di tipo Tram (flusso tramviario) se $s \in S_{Tram}$ e $d \in D_{Tram}$. Per ogni tipo di traffico (eccetto quello pedonale), possiamo quindi identificare l'insieme:

- $F_{TipoTraffico} = \{(s, d) \mid s \in S_{TipoTraffico}, d \in D_{TipoTraffico}\}$

Considerando l'unione dei vari insiemi $F_{TipoTraffico}$ possiamo definire l'insieme dei **flussi veicolari**, denotato dal simbolo:

- $F_v = \bigcup_{TipoTraffico\ \in T} F_{TipoTraffico}$

Definizione – Flusso pedonale

Un flusso (s, d) si dice **pedonale** se $s, d \in A$ e sono adiacenti ad una stessa strada che ammette attraversamento. L'insieme dei flussi pedonali di un incrocio è denotato dal simbolo:

- $F_p = \{(s, d) \mid s, d \in A \; e \; sono \; adiacenti \; a \; una \; stessa \; strada \; che \; ammette \; attraversamento\}$

Definizione – Cappio

Un flusso (s, d) si dice **cappio** se sia la sorgente che la destinazione appartengono alla stessa strada. L'insieme dei cappi è denotato dal simbolo:

- $F_c = \{(s, d) \mid s \; e \; d \; appartengono \; alla \; stessa \; strada\}$

Definizione – Flusso ammissibile

Un flusso (s, d) si dice **ammissibile** se la sorgente s e la destinazione d sono dello stesso tipo di traffico e appartengono a strade diverse. Sulla base delle definizioni precedenti si può dire, quindi, che un flusso è ammissibile se non è un cappio ed è un flusso veicolare o pedonale. L'insieme dei flussi ammissibili è denotato dal simbolo:

- $F_a = \left(F_v \cup F_p\right) \setminus F_c$

Definizione – Flusso vietato

L'utente può decidere, in fase di inserimento degli input al problema, di vietare il transito di veicoli in determinati flussi. Tali flussi vengono detti **vietati**. Individuiamo, quindi, il seguente insieme:

- $F_x = \{(s, d) \mid (s, d) \text{ è } vietato\}$

Definizione – Flusso ammesso

Un flusso (s, d) si dice **ammesso** se è un flusso ammissibile che non è stato esplicitamente vietato dall'utente. Definiamo quindi l'insieme dei flussi ammessi come:

- $F = F_a \setminus F_x$

Definizione – Semaforo

Ogni flusso ammesso è regolato da un semaforo. Ogni semaforo regola un solo flusso ammesso. Il semaforo può avere due stati: rosso e verde. Se il semaforo associato al flusso (s, d) è rosso nessun veicolo può percorrere il flusso (s, d). Se invece è verde, possono percorrerlo.

Collisioni tra flussi

Il problema principale che abbiamo dovuto risolvere è stata l'individuazione delle collisioni tra i vari flussi. Nello scheduling finale, infatti, non dovrà mai essere possibile che due flussi collidenti tra loro abbiano entrambi semaforo verde.

Definizione – Collisione

Dato un flusso (s, d) possiamo identificare quattro insiemi:

$LS_{(s,d)}$: l'insieme di tutte le sorgenti che si trovano a sinistra (left sources) rispetto al flusso (s, d);

$RS_{(s,d)}$: l'insieme di tutte le sorgenti che si trovano a destra (right sources) rispetto al flusso (s, d);

$LD_{(s,d)}$: l'insieme di tutte le destinazioni che si trovano a sinistra (left destinations) rispetto al flusso (s, d);

$RD_{(s,d)}$: l'insieme di tutte le destinazioni che si trovano a destra (right destinations) rispetto al flusso (s, d).

Un flusso (x, y) è in **collisione** con un altro flusso (s, d) se e solo se:

- $x \in LS_{(s,d)} \wedge y \in RD_{(s,d)}$
  oppure
- $x \in RS_{(s,d)} \wedge y \in LD_{(s,d)}$

Un flusso (x, y) è in **collisione debole** (**confluenza**) con un altro flusso (s, d) se e solo se:

- $x \neq s \wedge y = d$

Per chiarire meglio come funzionano le definizioni precedentemente illustrate proponiamo un esempio operativo, basato sulla figura 1.

Esempio

Con riferimento all'incrocio rappresentato nella figura 1, scegliamo ad esempio di calcolare tutti i flussi veicolari che collidono con il flusso (1,5), evidenziato in figura 2.

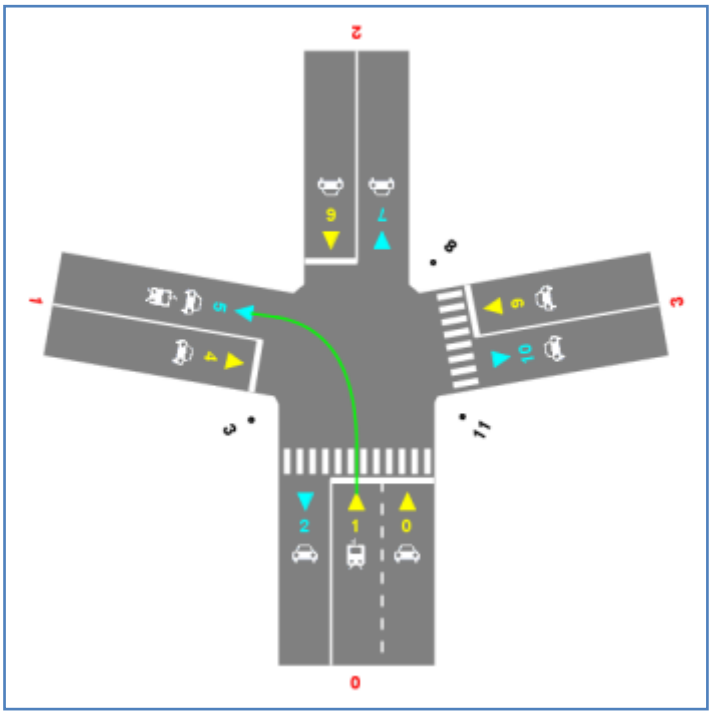


Figura 2 - Esempio numerico di flussi che collidono con (1,5)

Costruiamo gli insiemi definiti precedentemente:

- $S = \{0, 1, 4, 6, 9\}$
- $D = \{2, 5, 7, 10\}$
- $A = \{3, 8, 11\}$
- $LS_{(1,5)} = \{4\}$
- $RD_{(1,5)} = \{7, 10\}$
- $RS_{(1,5)} = \{0, 6, 9\}$
- $LD_{(1,5)} = \{2\}$

Seguendo la definizione di collisione possiamo ottenere la lista di tutti i flussi che collidono con (1,5), prendendo tutti i flussi (x, y) tali che:

- $x \in LS_{(1,5)} \ \wedge \ y \in RD_{(1,5)}$

oppure

- $x \in RS_{(1,5)} \ \wedge \ y \in LD_{(1,5)}$

Avremo, quindi i seguenti flussi collidenti:

- (4,7), (4,10) generati dalla prima condizione;
- (0,2), (6,2), (9,2) generati dalla seconda condizione.

Seguendo la definizione di confluenza possiamo ottenere la lista dei flussi confluenti con (1,5), prendendo tutti i flussi (x, y) tali che:

- $x \neq 1 \ \wedge \ y = 5$

Avremo, quindi i seguenti flussi confluenti:

- (0,5), (4,5), (6,5), (9,5)

La definizione di collisione appena descritta è piuttosto generale e, come si evince anche dall'esempio appena mostrato, soffre di due problemi:

1. nel calcolo delle collisioni non vengono considerati i flussi pedonali;
2. il calcolo delle collisioni genera anche flussi non ammissibili (cappi e flussi in cui sorgente e destinazione ammettono tipi di traffico diversi).

In realtà, come vedremo tra poco, è possibile ovviare a tali problemi utilizzando la definizione in modo intelligente ed eseguendo alcuni controlli aggiuntivi.

L'algoritmo che permette il calcolo delle collisioni, ovviando ai problemi precedentemente esposti, è il seguente:

Finché non sono stati considerati tutti i *flussi ammessi*:

1. si prende come riferimento un flusso ammesso che non sia già stato preso in precedenza (considerando anche i flussi pedonali, in quanto ammessi);
2. si utilizza la definizione di collisione per calcolare i flussi che collidono con quello di riferimento (qui invece, in base alla definizione, non vengono generati flussi pedonali);
3. ciascun flusso calcolato verrà considerato nel risultato solo se è ammesso.

Nell'algoritmo appena esposto, il fatto che i flussi pedonali possano essere presi come riferimento nel punto 1, ma non vengano considerati nel punto 2, potrebbe sembrare una limitazione. In realtà non è così per due motivi:

- due flussi pedonali non possono mai collidere tra loro, o meglio non vengono mai considerati collidenti;
- il concetto di collisione può essere visto come una *relazione simmetrica* tra due flussi e pertanto se esprimo il fatto che f1 *collide con* f2, non c'è bisogno di esplicitare anche che f2 *collide con* f1 (sarebbe inutilmente ridondante).

Per quanto riguarda i flussi veicolari, invece, il lettore attento potrebbe notare che la procedura precedentemente descritta genera vincoli ridondanti (a causa della simmetria della relazione di collisione). Proprio per questo il sistema prevede un'ulteriore ottimizzazione: quando il flusso preso come riferimento è veicolare, nel punto 2 dell'algoritmo non vengono generati i flussi presi come riferimento in precedenza, evitando così che ci siano ridondanze.

Come impostazione di default, il sistema esclude le confluenze dal calcolo delle collisioni. È comunque possibile, tramite una checkbox, includere le confluenze nel calcolo, ma ovviamente abilitando quest'opzione aumentano le collisioni totali, i vincoli risultanti sono più stringenti ed aumenta, quindi, la complessità del problema.

Traduzione in NuSMV

Per risolvere il problema dello scheduling, bisogna utilizzare un tipo di modellazione che permetta di rappresentare gli aspetti dinamici del sistema, legati al cambiamento dei semafori. Per tale motivo, è stato quindi necessario utilizzare la logica temporale. In particolare, abbiamo utilizzato il model checker NuSMV, che permette, negando i vincoli, di ottenere un controesempio che li soddisfi. Dopo aver specificato il modello precedentemente descritto, abbiamo proceduto alla sua traduzione in un formato leggibile da NuSMV. Il file NuSMV risultante è composto da un unico modulo (MODULE main). In tale modulo possiamo identificare due parti principali:

1. La sezione VAR
2. La sezione LTLSPEC

La sezione VAR contiene le variabili del problema. Nel nostro caso abbiamo deciso di definire, per ogni flusso ammesso, una variabile binaria f_i che rappresenta lo stato del semaforo associato a quel flusso. Il valore 0 rappresenta il fatto che il semaforo è rosso, mentre il valore 1 rappresenta il verde.

La sezione LTLSPEC contiene le formule in logica temporale che rappresentano i vincoli che devono essere rispettati dallo schedule risultante. Al fine di trovare uno schedule, è necessario negare l'AND delle formule che rappresentano i vincoli. NuSMV cerca di provare la validità della formula: se ci riesce allora significa che è impossibile trovare una soluzione che rispetti tutti i vincoli; se invece non ci riesce, allora NuSMV restituisce un controesempio. Tale controesempio è proprio uno schedule che rispetta tutti i vincoli specificati e quindi può essere preso come soluzione del problema. Per questo motivo, la cosa fondamentale è proprio la corretta traduzione dei vincoli (requisiti) in formule della logica temporale che andranno inserite nella sezione LTLSPEC. Come precedentemente descritto, i vari requisiti sono:

1. Ciascun semaforo che regola un flusso di traffico, deve diventare verde un numero infinito di volte (o, in altre parole, in ogni momento deve essere vero che prima o poi il semaforo diventerà verde).
2. Se due flussi semaforici collidono tra loro, allora non deve mai essere possibile che i loro semafori siano contemporaneamente verdi.
3. Ciascun semaforo deve rispettare gli eventuali vincoli di durata minima del verde, imposti dall'utente.
4. Ciascun semaforo deve rispettare gli eventuali vincoli di durata massima del rosso, imposti dall'utente.

Il requisito 1 comporta che per ogni flusso f_i deve essere presente una formula del tipo:

- $G\ F\ f_i$

Questo significa che in ogni istante deve essere vero che prima o poi la variabile f_i sia posta ad 1 (true), cioè il semaforo associato al flusso f_i sia verde.

Il requisito 2 è quello più importante perché quello relativo alle collisioni tra flussi. In particolare, per ogni flusso f_i sarà presente una formula che esprime il seguente concetto: in ogni istante, se il semaforo relativo al flusso f_i è verde, allora tutti i semafori associati ai flussi che collidono con f_i saranno rossi. Per mostrare un esempio, supponiamo che il flusso f_1 collida con i flussi f_4, f_5 e f_7. Avremo, dunque, il seguente vincolo:

- $G\ (f_1 \rightarrow !\, f_4\ \&\ !\, f_5\ \&\ !\, f_7)$

Nel sistema, quando vengono creati i vincoli di collisione, è stata implementata un'ottimizzazione per evitare di esprimere vincoli ridondanti. Per illustrare questa ottimizzazione, facciamo riferimento all'esempio appena visto, relativo alle collisioni del flusso f_1. Trasformando l'implicazione in una disgiunzione, il vincolo appena espresso può essere scritto come $G\ (\ (!\, f_1\ |\ !\, f_4)\ \&\ (!\, f_1\ |\ !\, f_5)\ \&\ (!\, f_1\ |\ !\, f_7)\ )$, dove ogni clausola con l'OR esprime la collisione tra due flussi. Consideriamo ora il flusso f_4 e supponiamo che collida con f_1 e f_6. Attenendoci alla definizione fornita, scriveremmo per f_4 un vincolo del tipo $G\ (f_4 \rightarrow !\, f_1\ \&\ !\, f_6)$, che equivale a $G\ (\ (!\, f_4\ |\ !\, f_1)\ \&\ (!\, f_4\ |\ !\, f_6)\ )$. Come si può vedere, la prima clausola con l'OR è uguale a quella presente nel vincolo scritto per f_1 e quindi posso anche evitarne la ripetizione, omettendola. Il vincolo generato dal sistema, seguendo questa ottimizzazione, sarà quindi semplicemente $G\ (f_4 \rightarrow !\, f_6)$. Se, procedendo con le varie ottimizzazioni, il termine a destra di qualche implicazione dovesse risultare vuoto, allora il vincolo verrebbe omesso completamente. Questa ottimizzazione è legata al fatto che il concetto di collisione può essere considerato come una relazione simmetrica tra flussi, come visto precedentemente nella sezione in cui è stato descritto il modello.

Il requisito 3 comporta che per ogni flusso f_i deve essere presente una formula per specificare la durata minima del verde del suo semaforo. In particolare la formula sarà presente solo se la durata minima è maggiore di 1. Un vincolo di durata minima pari a *n* istanti, genererà una formula che rappresenta il seguente concetto: in ogni istante, se il semaforo è rosso e diventa verde nell'istante successivo (primo istante verde), allora sarà verde anche negli n-1 istanti successivi al prossimo (per un totale appunto di almeno n istanti verdi). Maggiore sarà *n*, più stringente sarà il vincolo e maggiore sarà la complessità del problema. Per mostrare un esempio di formula, supponiamo che per il flusso f_i sia stato specificato dall'utente un vincolo di durata minima del verde pari a 3 istanti. La formula in logica temporale che esprime tale vincolo sarà:

- $G\ (!\, f_i\ \&\ X\ f_i \rightarrow X\ X\ f_i\ \&\ X\ X\ X\ f_i)$

Il requisito 4 comporta che per ogni flusso f_i deve essere presente una formula per specificare la durata massima del rosso del suo semaforo. Un vincolo di durata massima del rosso pari a *n* istanti, genererà una formula che rappresenta il seguente concetto: in ogni istante, se il semaforo è rosso ora e negli *n-1* istanti successivi, allora nell'*n*-esimo istante successivo sarà verde. Minore sarà *n*, più stringente sarà il vincolo e maggiore sarà la complessità del problema. Per mostrare un esempio di formula, supponiamo che per il flusso f_i sia stato specificato dall'utente un vincolo di durata massima del rosso pari a 5 istanti. La formula in logica temporale che esprime tale vincolo sarà:

- $G\ (!\ f_i\ \&\ X\ !\ f_i\ \&\ X\ X\ !\ f_i\ \&\ X\ X\ X\ !\ f_i\ \&\ X\ X\ X\ X\ !\ f_i \rightarrow X\ X\ X\ X\ X\ f_i)$

Esecuzione di NuSMV

Il file con la struttura appena descritta è dato in input al model checker NuSMV, al fine di trovare uno schedule che soddisfi tutti i vincoli. Fin dalle prime prove abbiamo notato che, in caso di presenza di vincoli sia di durata minima del verde che di durata massima del rosso, NuSMV, utilizzato con il symbolic model checking, impiega un tempo notevole e un gran quantitativo di memoria per eseguire la computazione. Addirittura, già con incroci di 4 strade e particolari vincoli di durata, NuSMV non riesce a completare la computazione poiché la quantità di memoria allocata dal programma supera il limite di 2 gigabyte. Non riuscendo a completare la computazione, abbiamo provato ad utilizzare il bounded model checking, con lunghezza di cammino sufficientemente grande (ad es. 40). Facendo varie prove ci siamo accorti che, euristicamente parlando, se c'è una soluzione, essa non ha mai un numero di stati eccessivamente elevato (nelle nostre prove è risultato sempre minore di 20), e comunque, anche se ci fosse, sarebbe una soluzione eccessivamente complicata (con troppi stati) e quindi di scarso interesse. Per tale motivo, possiamo ragionevolmente assumere che se il bounded model checking di lunghezza 30 o 40 non trova una soluzione, allora probabilmente non esiste nessuna soluzione al problema. Per lo stesso motivo, qualora si voglia utilizzare il symbolic model checking, si può ragionevolmente affermare che se entro un certo tempo la computazione non ha trovato una soluzione, allora probabilmente non ne esiste nessuna. Per supportare queste considerazioni, nel sistema è stata data la possibilità di scegliere tra symbolic e bounded model checking (nel caso di bounded si può specificare la lunghezza) e si può fissare una durata massima per la computazione (allo scadere di tale tempo, il processo che esegue la computazione viene terminato forzatamente).

Ovviamente, il motivo per cui a volte NuSMV non riesce a trovare la soluzione è che i vincoli specificati dall'utente sono troppo stringenti: essendoci molte collisioni, infatti, le durate minime del verde e quelle massime del rosso non potranno essere troppo vincolanti. Per risolvere questo problema, si possono seguire due strategie:

1. Rilassare i vincoli di durata.
2. Ridurre le collisioni tra i flussi.

La prima strategia è piuttosto semplice e consiste nel rilassamento dei vincoli di durata. Rilassare i vincoli significa ridurre la durata minima del verde e/o aumentare la durata massima del rosso. Nel nostro sistema si è scelto di lasciare tale strategia totalmente a carico dell'utente, che dovrà quindi lanciare nuovamente il programma, inserendo vincoli di durata meno stringenti.

Per la seconda strategia, invece, il discorso è più complicato. Anche in questo caso l'utente può agire di sua iniziativa aumentando il numero dei flussi vietati, oppure deselezionando, qualora l'avesse selezionata precedentemente (di default è già deselezionata), l'opzione che considera collisioni anche le confluenze. Oltre a questo, però, il sistema prevede anche un metodo di riduzione automatica delle collisioni, attraverso lo spezzamento di flussi (effettuato solo per i flussi auto).

Definizione – Spezzamento di un flusso

Dato un flusso (s, d), esso può essere spezzato in due flussi (s, a) e (b, d), dove a e b sono due corsie che appartengono ad una stessa strada di riferimento tale che:

- permetta l'inversione di marcia;
- sia situata a destra del flusso (s, d).

Per chiarire il concetto di spezzamento proponiamo la seguente figura, in cui il flusso (0,5) è stato sostituito dai due flussi (0,10) e (9,5).

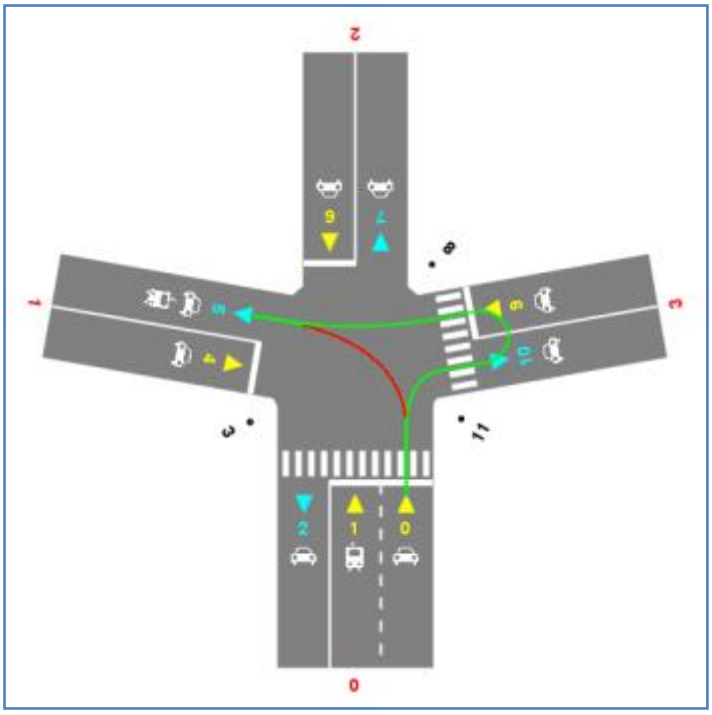


Figura 3 - Esempio di flusso spezzato

Lo spezzamento è stato possibile perché la strada 3 permette l'inversione di marcia e si trova a destra rispetto al flusso $(0{,}10)$.

Lo spezzamento elimina uno o più flussi e le collisioni relative ad essi e quindi semplifica il problema, aumentando la probabilità che esista una soluzione per esso. A seguito di uno spezzamento, il file NuSMV precedentemente generato viene modificato eliminando la variabile relativa al flusso spezzato e adattando opportunamente i vincoli che contenevano la variabile rimossa.

Nel sistema, vengono provati tutti i possibili spezzamenti di un flusso o di due flussi (vengono provate tutte le combinazioni possibili). Ci siamo fermati a due flussi poiché andando oltre il numero di casi possibili esploderebbe, rendendo la computazione troppo onerosa. Come ulteriore ottimizzazione, il sistema prova a spezzare i flussi in modo ordinato in base al numero di collisioni; cioè si prova prima a spezzare i flussi più

problematici e poi gli altri. Per una descrizione più approfondita del modo in cui vengono spezzati i flussi e dell'impatto sul file NuSMV risultante si rimanda alla sezione riguardante l'implementazione del sistema.

Implementazione Java

Il sistema JTrafficControl è implementato completamente in Java, un linguaggio Object Oriented scelto per via della sua nota versatilità e portabilità, che ci permette di eseguire il programma su diverse piattaforme hardware e software. In realtà, come abbiamo già detto in precedenza, oltre a Java è stato utilizzato anche NuSMV, un programma in grado di fare model checking su logiche temporali, prendendo in pasto file opportunamente formattati (file SMV) e restituendo in output il risultato della computazione. Il compito del programma Java è quello di permettere, attraverso una serie di schermate grafiche, la creazione dell'incrocio, la specifica dei vincoli da rispettare ed infine di generare i vari file SMV da usare come dati di input, per poi recuperare l'output generato e mostrare a video i risultati computati.

Il programma racchiude in sé numerosi algoritmi e classi necessarie per implementarli, che vengono richiamati in base alle esigenze di calcolo: basti pensare al fatto che il sistema prevede una serie di passaggi da rispettare prima di poter richiamare NuSMV ed ottenere l'output.

Il primo di questi passaggi è la creazione dell'incrocio, con la possibilità di inserire strade, corsie, tipologie di corsie, attraversamenti pedonali etc etc... Una volta definita la topologia dell'incrocio, bisogna numerare tutte le sorgenti e destinazioni dei possibili flussi e, per fare questo, viene richiamato l'algoritmo di numerazione. Lo scopo dell'algoritmo di numerazione è quello di preparare tutti i dati e di assegnare il corretto indice ad ogni sorgente e destinazione di un possibile flusso, dove per sorgente (o destinazione) intendiamo una corsia entrante (o uscente) ai quali bisogna aggiungere anche i punti di attraversamento pedonale. Solamente dopo aver eseguito l'algoritmo della numerazione è possibile specificare quali sono i flussi vietati (si possono vietare solamente flussi veicolari), indicando l'indice della sorgente e della destinazione vietata. Alla lista dei flussi vietati dall'utente va aggiunta anche la lista dei flussi banalmente non consentiti, detti anche cappi: un cappio non è altro che un flusso che ha come sorgente e destinazione, ovvero corsie, appartenenti alla stessa strada. Dopo aver raccolto queste informazioni il programma può generare la lista di tutti i flussi permessi, che verranno considerati per la generazione del file SMV. Su questa lista possono essere applicati dei vincoli, di modo che siano rispettate delle opportune proprietà, come magari la durata massima del rosso per gli attraversamenti pedonali, o la durata minima del verde per flussi auto e così via. Questi vincoli possono essere omessi, oppure specificati per ogni tipologia di flusso (pedonale, auto, tramviario) o ancora specificati per ogni singolo flusso, cosicché si possa garantire il più alto livello di personalizzazione e generalizzazione del comportamento dell'incrocio. Solo dopo esser arrivati a questo punto è possibile avviare la computazione di NuSMV, in quanto il programma Java si prende l'incarico di generare il file SMV e lo invia all'eseguibile, aspettando il risultato di output.

Il programma Java riesce a gestire anche un timeout di esecuzione, passato il quale abortisce la computazione di NuSMV, senza aspettare che questo ritorni un risultato, positivo o negativo che sia. Questa funzionalità si è resa necessaria in quanto si è verificato empiricamente che già a partire da incroci modesti, il tempo di calcolo necessario a NuSMV per ritornare un risultato può essere elevato ed oneroso, rendendo di fatto poco interessante il calcolo stesso.

Difatti, se NuSMV ritorna l'insoddisfacibilità per le formule generate per risolvere il modello, o se viene bloccato prima a causa del timer, il programma provvede subito a spezzare il flusso con il maggior numero di collisioni (come evidenziato nella spiegazione del modello) provando tutte le possibili combinazioni finché non viene trovato uno schedule soddisfacente. Se anche in questo modo la soluzione non dovesse essere trovata, allora viene spezzato un ulteriore flusso e generate nuovamente tutte le possibili

combinazioni. Questa procedura di spezzare i flussi con il maggior numero di collisioni viene però fermata con il limite dei due flussi, altrimenti ci troveremmo di fronte ad una esplosione delle combinazioni, che di fatto renderebbero ingestibile la computazione.

Data la diversa natura delle componenti del programma, abbiamo sviluppato l'applicazione suddividendo le varie classi in 4 differenti package di raccolta:

- **Grafica**: in questo package sono contenute tutte le classi che si occupano di gestire le finestre dell'applicazione. Sono altresì presenti le classi che si occupano della generazione del disegno dell'incrocio.
- **Motore**: in questo package sono contenute tutte le classi che in qualche modo si occupano della logica applicativa. Troviamo quindi la classe che genera i file SMV, così come la classe che controlla l'esecuzione di NuSMV e quella che effettua il parsing dell'output generato.
- **Struttura**: in questo package sono memorizzate le classi utilizzate per mantenere in memoria tutte le informazioni necessarie per la caratterizzazione di un incrocio.
- **Icone**: in questo package sono raccolte tutte le icone utilizzate nell'interfaccia grafica del programma.

Struttura

In questo package sono presenti 7 classi Java che ci permettono di memorizzare tutte le informazioni necessarie per caratterizzare completamente un incrocio. Un incrocio è memorizzato come un *Vector* (praticamente una lista) di strade, le quali a loro volta hanno un *Vector* per le corsie entranti ed un altro *Vector* per le corsie uscenti. Ogni corsia può essere solamente di tipo entrante o uscente ed è associata ad una o più tipologie di traffico con il vincolo che, se si tratta di una corsia entrante, è possibile specificare solamente una sola tipologia di traffico. A sua volta ogni strada non può avere più corsie dello stesso tipo ovvero una strada non può avere, per esempio, due corsie entranti adibite a traffico di automobili (cfr. la sezione sulla spiegazione del modello).

La classi implicate nella realizzazione di questa struttura sono le seguenti:

- ***TipoTraffico.java***: in questa classe sono definiti i valori che identificano i vari tipi di traffico. Il sistema permette di gestire corsie con traffico di automobili, corsie adibite al passaggio di tram ed infine mantiene traccia anche del traffico pedonale, riservando a quest'ultimo un particolare valore.
- ***Corsia.java***: in questa classe vengono memorizzate tutte le informazioni di una corsia. Praticamente viene tenuta traccia del tipo di corsia (entrante o uscente); qual è l'indice della corsia nella propria strada di appartenenza (questa informazione è utile ai fini dell'algoritmo di numerazione); il riferimento all'oggetto Strada ed infine un *Vector* con i *TipoTraffico* associati alla corsia, con il vincolo che in caso di corsia entrante è possibile specificare una sola tipologia di traffico.
- ***Strada.java***: in questa classe vengono memorizzate tutte le informazioni necessarie per caratterizzare una strada. La classe ha due *Vector*, uno utilizzato come lista di corsie entranti e l'altro utilizzato per le corsie uscenti. Ognuno di questi *Vector* non è altro che una lista di oggetti di tipo *Corsia*. Inoltre ogni oggetto istanza della classe *Strada* tiene traccia della possibilità di permettere attraversamento pedonale, di garantire l'inversione di marcia (informazione utilizzata

nell'algoritmo che spezza i flussi) ed infine memorizza l'indice della strada nell'incrocio (praticamente l'indice della posizione nel *Vector* dell'incrocio).

- ***Incrocio.java***: questa classe mantiene tutta l'informazione sull'incrocio, memorizzando il tutto attraverso un *Vector* di oggetti Strada. La classe è implementata attraverso il pattern *Singleton*, perché l'istanza da utilizzare in tutte le classi del programma deve essere sempre la medesima (si lavora sempre sopra lo stesso incrocio).

Un altro particolare importante, riguardante queste 4 classi, è quello dell'implementazione dell'interfaccia *Serializable*, che ci permette di salvare (serializzare) facilmente su disco, dando quindi la possibilità all'utente di salvare una tipologia di incrocio e riprenderla in seguito per effettuare altre prove.

Una rappresentazione grafica della struttura utilizzata è visibile alla figura 4, presentata poco più avanti nel testo.

Per completare la descrizione mancano da analizzare le ultime 3 classi, che entrano in gioco solamente dopo l'esecuzione dell'algoritmo di numerazione:

- ***Flusso.java***: questa classe permette di descrivere un flusso, così come è stato definito precedentemente nel modello. Un flusso è caratterizzato dall'indice della *Corsia* sorgente, della *Strada* sorgente, della *Corsia* di destinazione e della *Strada* di destinazione, ai quali vanno aggiunte le informazioni riguardanti i vincoli sul flusso, ovvero la durata massima del rosso, la durata minima del verde e la tipologia del flusso (auto o tram).
- ***ListaFlussi.java***: questa classe permette di mantenere una lista di flussi attraverso la creazione di un *Vector* di oggetti Flusso. I servizi offerti da *ListaFlussi* verranno utilizzati nel package motore per mantenere informazioni sui flussi vietati, sui flussi pedonali e sui flussi che rappresentano cappi e così via.
- ***ArrayOrdinatoCircolare.java***: questa classe java è stata creata appositamente per tenere in memoria una struttura dati di appoggio in grado di facilitare alcune operazioni necessarie all'algoritmo per determinare i flussi e le collisioni fra questi.

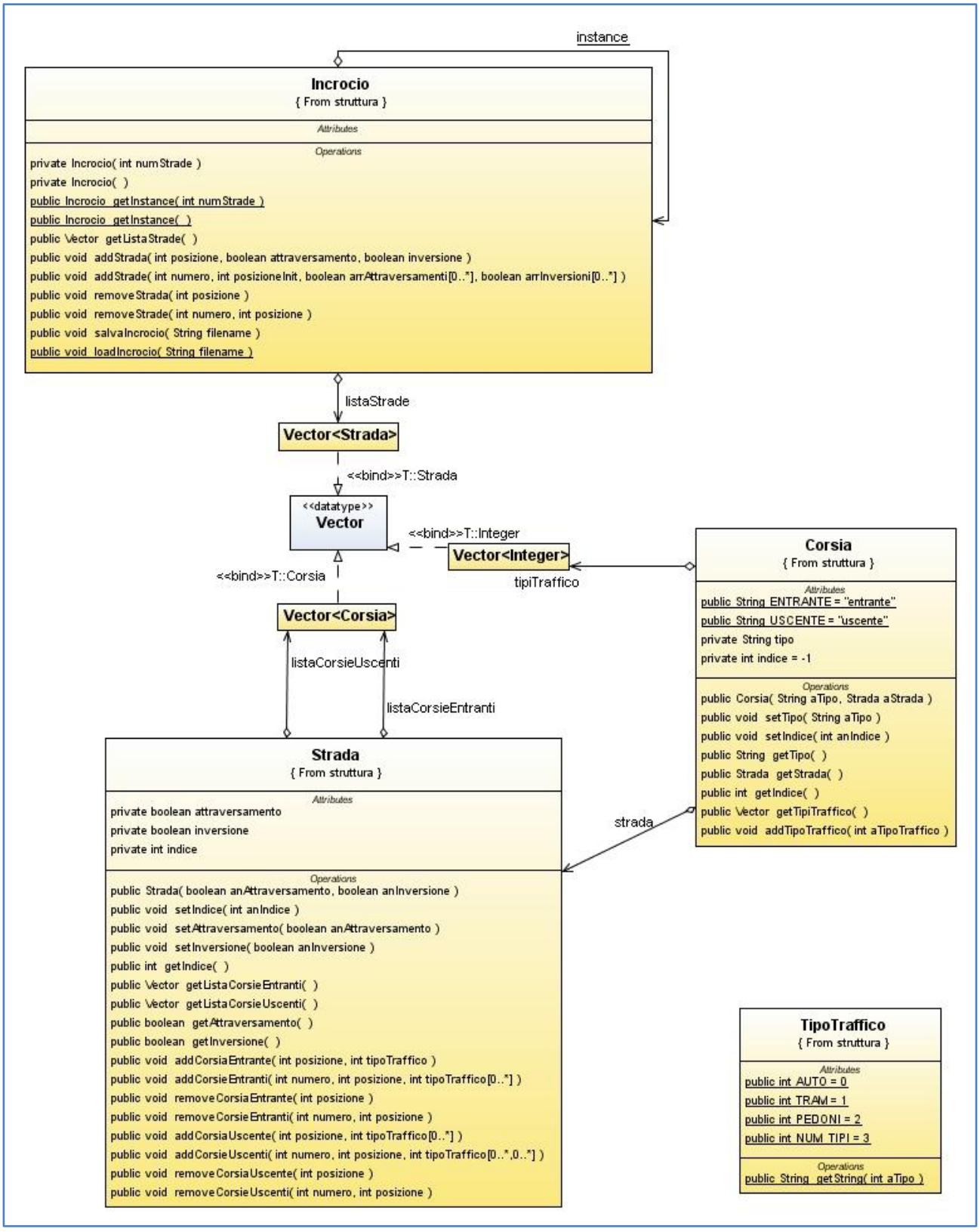


Figura 4 – Diagramma UML delle classi relative al mantenimento delle informazioni sull'incrocio.

Motore

In questo package sono presenti 5 classi, che si occupano principalmente di gestire alcuni degli algoritmi già citati precedentemente, nonché di preparare i file di input per NuSMV ed effettuare il parsing dell'output generato.

Le classi sono:

- ***Launcher.java***: questa classe si occupa principalmente di gestire l'interazione tra Java e NuSMV. Permette di richiamare NuSMV passandogli alcuni parametri di esecuzione (come per esempio la possibilità di utilizzare il Bounded Model Checking), nonché di controllarne l'esecuzione e recuperare l'output generato.
- ***LauncherManager.java***: questa classe implementa l'interfaccia Runnable di Java perché mantiene in un thread apposito il controllo delle computazioni di NuSMV da mandare in esecuzione. Il sistema in effetti mantiene dentro una struttura dinamica (Vector) tutte le richieste NuSMV da mandare in esecuzione, testando continuamente, con un thread apposito, questa struttura dati. Il controllo della struttura dati è affidata ad un thread per poter liberare le risorse della macchina virtuale java per il disegno degli oggetti swing di modo che, in caso di esecuzione di NuSMV, l'applicazione grafica non risulti bloccata. Ogni qualvolta viene aggiunta una richiesta nella struttura dati questa prima o poi verrà processata da un altro thread, cosicché NuSMV in realtà viene eseguito in un thread separato rispetto a quello che effettua il monitoraggio della struttura dati. Quest'ultimo thread lancia al massimo una esecuzione di NuSMV alla volta, per non saturare il processore con troppi processi NuSMV, notoriamente avidi di risorse. Il rilancio di un nuovo processo NuSMV viene effettuato solamente quando la computazione precedente è terminata.
- ***KillProcess***.java: questa classe estende la classe TimerTask e permette di gestire il timer associato ad ogni computazione di NuSMV. Il valore numerico associato a questo timeout è configurabile dall'interfaccia grafica e permette, allo scadere del tempo impostato, di bloccare coattivamente la computazione di NuSMV e se presente, lanciare una nuova computazione.
- ***VarData.java***: questa classe è stata creata unicamente per memorizzare correttamente i valori di output restituiti da NuSMV. Quando NuSMV ritorna un controesempio di soddisfacibilità del modello, mostra in output il nome della variabile ed il suo valore[2]. Questa classe non è altro che una struttura dati creata *ad hoc* per memorizzare queste due informazioni (nome variabile e valore associato).
- ***Parser.java***: questa classe Java analizza la stringa di output generata da NuSMV, andando a verificare se il programma ha generato o meno un controesempio numerico (il nostro schedule cercato). In caso di controesempio, la classe Parser offre un metodo per effettuare il parsing e creare una struttura dati di *VarData*, idonea per memorizzare le informazioni.
- ***JavaToNuSMV.java***: questa è sicuramente la classe portante di tutta la tesina. Entrando nello specifico questa classe si occupa di mantenere informazioni sui flussi vietati, sui flussi pedonali, sui cappi, sugli indici delle sorgenti, delle destinazioni e degli attraversamenti pedonali. Invece tra gli algoritmi implementati c'è quello già citato della numerazione; quello per creare la lista dei flussi da considerare[3]; l'algoritmo che associa ad ogni flusso le strade interessate; l'algoritmo che verifica il flusso con il maggior numero di collisioni e lo spezza in due flussi; l'algoritmo che verifica la

[2] In realtà NuSMV ritorna il valore della variabile solamente quando questa subisce un cambiamento di stato, ovvero quando il valore associato viene modificato.
[3] I flussi da considerare ai fini del modello sono tutti i flussi ammessi.

creazione di tutti i modelli possibili data la necessità di spezzare i flussi ed infine l'algoritmo più importante di tutti, che traduce l'incrocio con tutti i suoi vincoli in un file SMV da dare in pasto a NuSMV. Data la complessità della classe, e data la natura dei dati gestiti, la classe è stata implementata utilizzando il pattern *Singleton*.

Algoritmo per la numerazione

L'algoritmo di numerazione (realizzato nella funzione *numerazione()* contenuta nella classe JavaToNuSMV), preleva l'istanza dell'oggetto contenente la rappresentazione dell'incrocio, ne analizza la topologia e numera le corsie e gli altri punti di interesse dell'incrocio in base al modello descritto nelle sezioni precedenti. Contestualmente a ciò, vengono anche predisposte e popolate varie strutture dati, necessarie per i passi successivi della computazione. La funzione effettua due visite della lista delle strade dell'incrocio.

Nella prima visita, il sistema utilizza un contatore (inizializzato a 0) per numerare i vari punti di interesse dell'incrocio. In particolare, per ogni strada, vengono numerate prima le corsie entranti, poi quelle uscenti e infine l'eventuale punto di attraversamento posto subito dopo la strada. Un punto di attraversamento viene inserito dopo una strada se e solo se la strada stessa o quella successiva hanno il flag di attraversamento posto a true. Durante questa prima visita, inoltre, vengono costruite e popolate le seguenti strutture dati:

- **strade** – è un ArrayOrdinatoCircolare contenente gli indici di tutte le strade;
- **sorgenti** – è un array di oggetti ArrayOrdinatoCircolare, indicizzato in base ai tipi di traffico. Ciascun oggetto *sorgenti[TipoTraffico]* contiene gli indici delle corsie entranti che permettono il transito di veicoli di tipo TipoTraffico.
- **destinazioni** – è un array di oggetti ArrayOrdinatoCircolare, indicizzato in base ai tipi di traffico. Ciascun oggetto *destinazioni[TipoTraffico]* contiene gli indici delle corsie uscenti che permettono il transito di veicoli di tipo TipoTraffico.
- **attraversamenti** – è un ArrayOrdinatoCircolare contenente tutti gli indici dei punti di attraversamento pedonale.
- **flussiCappi** – è un oggetto di tipo ListaFlussi che contiene tutti i flussi che hanno sorgente e destinazione appartenenti alla stessa strada.

La seconda visita, invece, serve per costruire e popolare la struttura dati **flussiPedonali**, di tipo ListaFlussi, contenente tutti i flussi pedonali ammessi dall'incrocio: per ogni strada che ha il flag di attraversamento settato a true, viene inserito in flussiPedonali il flusso che ha come sorgente il punto di attraversamento immediatamente precedente alla strada e come destinazione quello immediatamente successivo.

Algoritmo per la creazione della lista dei flussi

L'algoritmo di creazione della lista dei flussi (realizzato nella funzione *creaListaFlussi()* contenuta nella classe JavaToNuSMV), ha come obiettivo la creazione della lista dei soli flussi ammessi, che verrà memorizzata nella struttura dati **listaFlussi**, di tipo ListaFlussi. Seguendo le definizioni descritte nelle sezioni precedenti, l'algoritmo svolge le seguenti azioni per ottenere la lista dei flussi ammessi:

- inserisce in listaFlussi tutti i flussi veicolari, cioè i flussi che hanno sorgente e destinazione dello stesso tipo di traffico; per creare tali flussi vengono utilizzate le strutture dati *sorgenti* e *destinazioni*, costruite nella funzione di numerazione;

- sottrae a listaFlussi i flussi vietati dall'utente, contenuti nella lista flussiVietati, usando la primitiva subtractListaFlussi() prevista dalla classe ListaFlussi;
- sottrae a listaFlussi anche i cappi, contenuti nella lista flussiCappi;
- aggiunge a listaFlussi i flussi pedonali, contenuti nella lista flussiPedonali.

Algoritmo per la creazione del file NuSMV

Questo algoritmo (realizzato nella funzione *creaFileNuSMV()* contenuta nella classe JavaToNuSMV) utilizza le varie strutture dati costruite precedentemente per creare il codice NuSMV che rappresenta l'incrocio con tutti i suoi vincoli.

Il file inizia con la dicitura `MODULE main` che sta ad indicare l'inizio dell'unico modulo che compone il file, denominato appunto *main*.

Dopodiché viene costruita la sezione VAR, contenente la dichiarazione di tutte le variabili utilizzate dal modulo. Come spiegato in precedenza, viene dichiarata una variabile booleana per ogni flusso ammesso. I nomi delle variabili vengono assegnati a ciascun flusso dalla classe ListaFlussi che ospita i flussi ammessi (tramite la funzione *getFlussoVarName()*), e sono del tipo f0, f1, f2, ecc.

Per il flusso f0 = (0,5), ad esempio, ci sarà la seguente riga nel file NuSMV:

```
f0: boolean; -- (0,5)
```

Dopo la definizione delle variabili, c'è la sezione LTLSPEC, contenente tutti i vincoli dell'applicazione. Come accennato precedentemente, per usare NuSMV come solutore è necessario negare l'AND di tutti i vincoli del problema. La sezione LTLSPEC generata avrà la seguente struttura:

```
LTLSPEC
!(
      -- Formule che indicano che prima o poi un semaforo deve diventare verde
            vincolo &
            vincolo &
            ecc…
      -- Formule che evitano le collisioni
            vincolo &
            ecc…
      -- Formule che esprimono la durata minima del verde
            vincolo &
            ecc…
      -- Formule che esprimono la durata massima del rosso
            vincolo &
            ecc…
```

)

I singoli vincoli vengono generati seguendo le regole e le ottimizzazioni descritte precedentemente nella sezione relativa alla traduzione in NuSMV. In particolare:

- I vincoli che indicano che in ogni istante prima o poi un semaforo deve diventare verde, sono di banale costruzione (vedi sezione relativa alla traduzione in NuSMV).
- I vincoli di collisione vengono costruiti applicando le definizioni viste nella sezione relativa alla descrizione del modello, considerando solo i flussi ammessi (cioè quelli contenuti in listaFlussi). Basandosi sugli insiemi *sorgenti* e *destinazioni*, vengono creati i quattro insiemi necessari alla specifica delle collisioni, grazie alla primitiva *getSubset()* della classe ArrayOrdinatoCircolare. Nella creazione dei vincoli vengono applicate le ottimizzazioni descritte nella sezione relativa alla traduzione in NuSMV.
- I vincoli di durata minima del verde e di durata massima del rosso vengono creati sulla base delle varie durate specificate dall'utente in fase di input, contenute all'interno degli oggetti rappresentanti i singoli flussi.

Algoritmo per la sostituzione dei flussi (spezzamenti)

Questo algoritmo (realizzato nella funzione *sostituisciFlusso()* contenuta nella classe JavaToNuSMV) si occupa di generare le sostituzioni relative a tutti i possibili spezzamenti di uno o due flussi.

L'algoritmo comincia a calcolare tutti i possibili spezzamenti di un solo flusso. A tale scopo, i flussi vengono ordinati in base al numero di collisioni che hanno con gli altri. Per ogni flusso (s,d), partendo da quello con più collisioni e procedendo in ordine decrescente di collisioni (saltando i flussi che non generano nessuna collisione e che quindi non hanno motivo di essere spezzati), l'algoritmo calcola le strade candidate, cioè quelle che ammettono inversione e che si trovano a destra del flusso. Per ogni strada candidata si prende la corsia uscente di tipo auto (u) e la corsia entrante di tipo auto (e). Se i flussi (s,u) e (e,d) sono ammessi (cioè sono contenuti in *listaFlussi*) genero una sostituzione che pone (s,d)=(s,u)+(e,d). Supponendo che il file generato senza spezzamenti sia "Incrocio.smv" e le variabili associate a (s,d), (s,u) e (e,d) siano rispettivamente f0, f1 ed f2, l'algoritmo invocherà la funzione di modifica del file NuSMV con tali parametri e lancerà l'esecuzione di NuSMV passandogli il nuovo file creato ("Incrociof0f1f2.smv" – vedi l'algoritmo per la modifica del file NuSMV). Se non si ottiene soluzione, allora l'algoritmo prova con un'altra sostituzione (si considera un'altra strada candidata se ce ne sono ancora, altrimenti si prova a spezzare un altro flusso). Se invece si ottiene una soluzione l'algoritmo termina. Ciascuna sostituzione generata viene rappresentata con una tripla (ad esempio [f0, f1, f2]) espressa come array di 3 stringhe. Tali triple vengono inserite, man mano che sono generate, in un vettore chiamato *combinazioniFlussi*.

Se, provando tutte le possibili sostituzioni generate dagli spezzamenti di un solo flusso non si ottiene nessuna soluzione, l'algoritmo prova a spezzare due flussi contemporaneamente. Utilizzando il vettore *combinazioniFlussi*, l'algoritmo genera tutte le possibili coppie di sostituzioni senza doppioni, cioè se ha già generato la coppia di triple (t1,t2), allora non considererà (t2,t1). Per ogni coppia generata, l'algoritmo verifica che sia una coppia compatibile, cioè che i flussi sostituiti tramite una tripla non vengano a loro volta spezzati dall'altra tripla. Questo controllo è eseguito dalla funzione *verificaCompatibilita()* che viene invocata su ciascuna coppia generata. Se la coppia è compatibile, allora si procede con la modifica del file NuSMV e con l'esecuzione. In particolare, supponendo che la coppia sia ([f0,f1,f2], [f3,f4,f5]), verrà invocata la funzione di modifica del file NuSMV con parametri "Incrociof0f1f2.smv" (che è il file in cui si applica solo

la prima sostituzione), f3, f4 e f5. Il file risultante ("Incrociof0f1f2f3f4f5.smv") verrà dato in pasto a NuSMV per cercare una soluzione. Anche in questo caso, se il solutore trova uno schedule l'algoritmo si ferma, altrimenti si prova con un'altra coppia. Se considerando tutte le coppie non trovo nessuna soluzione l'algoritmo si ferma, poiché procedere alle sostituzioni di tre flussi sarebbe troppo oneroso.

Algoritmo per la modifica del file NuSMV

Ogni volta che viene generata una sostituzione, relativa ad uno spezzamento, deve essere creato un nuovo file da dare in pasto a NuSMV per verificare se ci sia una soluzione o meno. La creazione di un file NuSMV ex-novo, comporterebbe la necessità di ricalcolare ed aggiornare tutte le strutture dati precedentemente descritte per adeguarle allo spezzamento, dopodiché si dovrebbe procedere nuovamente alla costruzione del codice NuSMV. Invece di adottare questo approccio piuttosto oneroso, abbiamo preferito creare il nuovo file prendendo come base il file creato precedentemente ed applicando le modifiche, causate dallo spezzamento, direttamente sul codice NuSMV. Questo metodo, oltre ad essere notevolmente più semplice, rende anche molto facile lo spezzamento di ulteriori flussi. Se infatti vogliamo spezzare due flussi contemporaneamente, basta prendere come base il file, generato precedentemente, che considera lo spezzamento di uno solo dei due flussi, e applicare su di esso le modifiche relative allo spezzamento dell'altro.

L'algoritmo per la modifica del file NuSMV (realizzato nella funzione *modificaFileNuSMV()* contenuta nella classe JavaToNuSMV), prende come input il nome di un file NuSMV precedentemente generato, il nome della variabile che rappresenta il flusso da spezzare e i nomi delle variabili dei due flussi risultato dello spezzamento. L'obiettivo dell'algoritmo è di creare un nuovo file NuSMV che modifichi quello vecchio rappresentando lo spezzamento del flusso specificato. Il nuovo file creato avrà un nome dato dal nome di quello vecchio più i nomi dei tre flussi coinvolti nello spezzamento. Ad esempio se il nome del file originario è "Incrocio.smv" e lo spezzamento in questione è f0 = f1+f2, il nome del nuovo file sarà "Incrociof0f1f2.smv".

L'algoritmo legge il file originario riga per riga e applica i cambiamenti necessari ad esprimere lo spezzamento. Tali cambiamenti sono:

nella sezione VAR

- viene rimossa la riga che dichiara la variabile relativa al flusso da spezzare;

nella sezione LTLSPEC

- nella riga contenente il vincolo che esprime che il semaforo relativo al flusso da spezzare prima o poi deve diventare verde, la variabile del flusso da spezzare viene sostituita con l'AND delle variabili dei due flussi risultato;
- vengono rimosse le righe relative ai vincoli di collisione in cui il flusso da spezzare appare nel termine a sinistra dell'implicazione oppure appare come unico letterale presente nel termine a destra;
- in ogni altro vincolo di collisione che non ricade nel caso precedente viene rimosso dal termine a destra dell'implicazione qualsiasi riferimento alla variabile del flusso da spezzare;
- nella riga contenente il vincolo di durata minima del verde relativo al flusso da spezzare, la variabile del flusso da spezzare viene sostituita con l'AND delle variabili dei due flussi risultato;

- nella riga contenente il vincolo di durata massima del rosso relativo al flusso da spezzare, la variabile del flusso da spezzare viene sostituita con l'AND delle variabili dei due flussi risultato.

Grafica

Il package grafica raccoglie 20 classi che implementano tutto l'aspetto grafico della tesina. Nello specifico le classi sono:

- ***Main.java***: è la classe Java che implementa il metodo *main*. E' la classe di partenza dell'applicazione.
- ***Principale.java***: è la classe che implementa il JFrame principale dell'applicazione. Oltre a questo, la classe si preoccupa anche di caricare i giusti pannelli, considerando l'interazione di tutto il sistema. In pratica permette di cambiare al JFrame il contentPane, passando da PrimoPassoPanel a SecondoPassoPanel, TerzoPassoPanel ed infine QuartoPassoPanel.
- ***PrimoPassoPanel.java***: questa classe implementa il pannello presentato a video durante il primo passo dell'interazione. Attraverso questo pannello è possibile aggiungere (eliminare) strade all'incrocio, nonché visualizzare il disegno dell'incrocio risultante.
- ***Disegno.java***: questa classe implementa un pannello Java progettato appositamente per disegnare l'incrocio. Ogni qual volta viene aggiornata la struttura dati dell'incrocio, viene richiamato il disegno, che aggiorna la visualizzazione a video dell'incrocio. Il pannello riesce a gestire un numero arbitrario di strade, adattando le dimensioni del disegno al numero di strade ed al numero di corsie di ogni strada.
- ***StradaForma.java***: in questa classe sono memorizzate le istruzioni necessarie per disegnare una singola strada. Viene richiamata per ogni strada da disegnare da *Disegno.java*, la quale organizza i disegni di modo che non si sovrappongano.
- ***AggiungiStradaFrame.java***: questa classe implementa il frame che si occupa di aggiungere una strada all'incrocio. Per ogni strada è possibile aggiungere (o eliminare) corsie entranti o uscenti, nonché selezionare se è previsto un attraversamento pedonale e se è prevista la possibilità di effettuare inversione, e quindi di usare la strada per spezzare un flusso. Ovviamente non è possibile effettuare l'inversione per una strada a senso unico, ovvero con corsie dello stesso tipo, oppure strade che hanno solo corsie di tipo tramviario.
- ***EliminaStradaFrame.java***: questa classe implementa il frame che permette di disegnare la finestra attraverso la quale si possono eliminare strade dall'incrocio. Dato che le strade vengono presentate attraverso una combo box, non è possibile eliminare strade che non esistono, minimizzando la possibilità di errore.
- ***AggiungiCorsiaFrame.java***: questa classe implementa il frame che si occupa di aggiungere una corsia alla strada selezionata. E' previsto un controllo per cui non è possibile specificare più di una tipologia di traffico per le corsie entranti (questo non è vero per le corsie uscenti).
- ***EliminaCorsiaFrame.java***: questa classe implementa il frame che si occupa della finestra in grado di gestire l'eliminazione di una corsia da una strada.
- ***SecondoPassoPanel.java***: questa classe realizza il pannello necessario per stampare a video il secondo passo dell'interazione con il sistema, ovvero quando si debbono scegliere i flussi vietati. Dall'interfaccia è possibile aggiungere o eliminare un flusso vietato, nonché controllare la lista dei flussi già dichiarati vietati, tramite una comoda textarea.
- ***AggiungiFlussoVietatoFrame.java***: questa classe implementa il frame dedicato per l'aggiunta di un flusso vietato all'incrocio. Per aggiungere un flusso vietato vengono presentate due combo box:

una per la sorgente del flusso ed una per la destinazione. Ovviamente viene fatto un controllo affinché la sorgente e la destinazione siano riferite alla stessa tipologia di traffico. Non è possibile vietare un flusso pedonale.

- ***EliminaFlussoVietatoFrame.java***: questa classe realizza il frame dedicato all'eliminazione di un flusso vietato precedentemente creato. La scelta del flusso da eliminare è mediata da una combo box, motivo per il quale non è possibile eliminare un flusso vietato non specificato.
- ***TerzoPassoPanel.java***: questa classe è responsabile del pannello dedicato al terzo passo dell'interazione, quando vengono stabiliti i vincoli temporali sui flussi ammessi nella computazione. E' possibile stabilire nessun vincolo temporale, oppure stabilire vincoli per ogni tipologia di traffico (auto, tram e pedoni), oppure stabilire i vincoli per ogni flusso specifico. In questa schermata è anche possibile decidere se considerare o meno le confluenze, ovvero flussi che hanno la medesima destinazione, come collisioni.
- ***AggiungiVincoloFlussoFrame.java***: questa classe implementa il frame responsabile di aggiungere vincoli temporali su un singolo flusso. Il flusso viene scelto da un elenco tramite combo box e attraverso due textfield si possono specificare i valori per la durata minima del verde e la durata massima del rosso.
- ***EliminaVincoloFlussoFrame.java***: questa classe realizza il frame responsabile della finestra che permette di eliminare vincoli temporali, precedentemente specificati, da un determinato flusso.
- ***NonPiuDiDueCifre.java***: questa classe estende una *JTextField*, personalizzandola di modo che accetti solamente cifre numeriche, che in questo caso sono limitate a 2 cifre.
- ***ContaCaratteri.java***: questa classe non è altro che la dichiarazione di una interfaccia, utilizzata a sua volta da *NonPiuDiDueCifre.java*.
- ***QuartoPassoPanel.java***: questa classe implementa l'ultimo pannello del programma, dove si può lanciare la computazione e verificare l'output di NuSMV. L'output può essere verificato sia sotto forma di tabella, sia sotto forma di grafico, evidenziando i flussi direttamente sul disegno dell'incrocio.
- ***JCurve.java***: questa classe è la responsabile per il disegno dei flussi abilitati istante per istante. Lo scopo è quello di tracciare un arco, o una retta, tra la sorgente e la destinazione.
- **ExtFilter.java**: questa classe crea un filtro da associare alla finestra di selezione dei file. In pratica viene utilizzata per filtrare i soli file inc, per aprire incroci precedentemente salvati.

Esempio di utilizzo

La schermata principale del programma è composta da un menu e due pannelli, che rispettivamente servono per disegnare e per personalizzare l'incrocio.

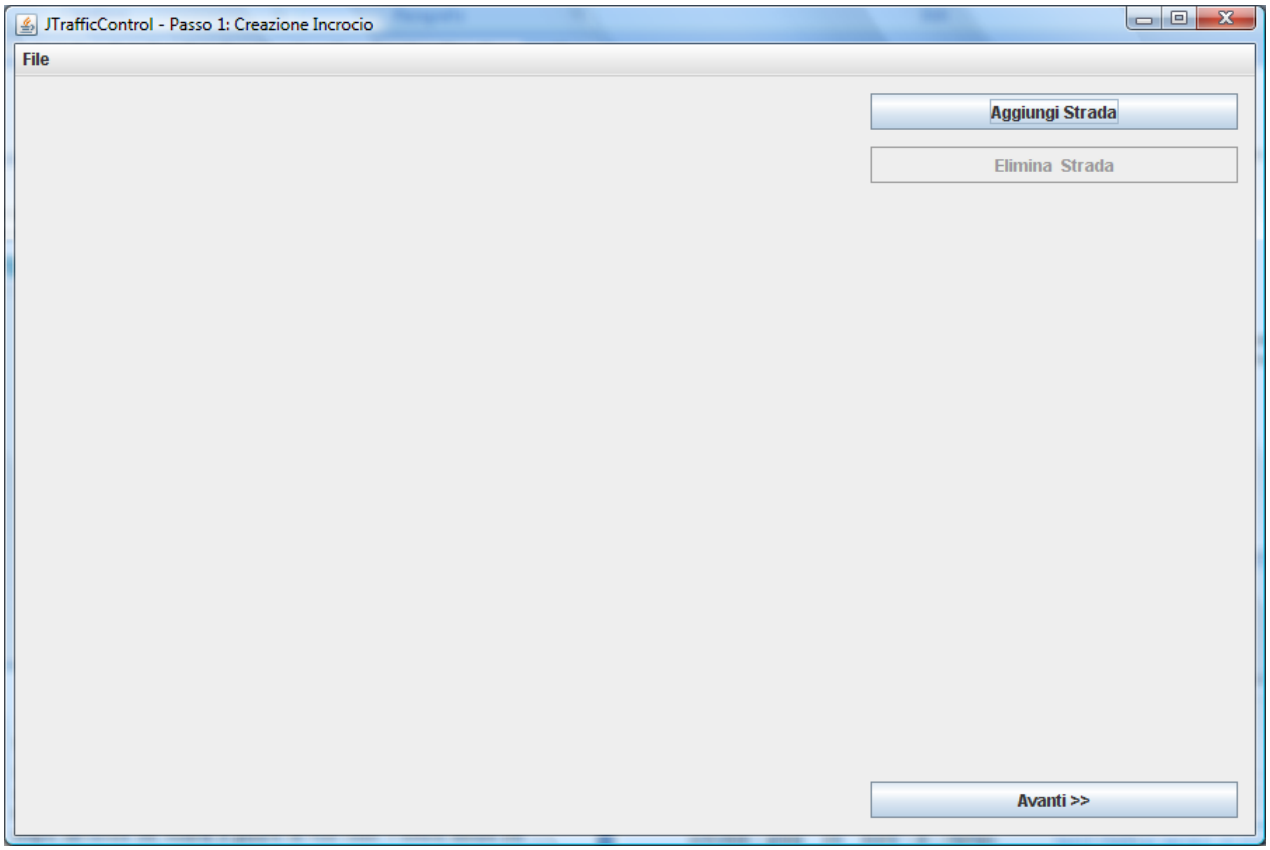


Figura 5- Schermata principale di JTrafficControl

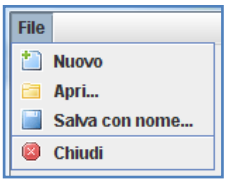


Figura 6 - Menu File

Attraverso il menu File è possibile iniziare con un nuovo lavoro, aprire un incrocio precedentemente salvato, salvare l'incrocio creato ed infine chiudere il programma.

Nel pannello laterale di destra si trovano invece 3 bottoni che hanno il compito di aggiungere una strada all'incrocio, eliminare una strada dall'incrocio ed infine l'ultimo bottone ha il compito di andare avanti con l'esecuzione e passare dalla modalità di disegno dell'incrocio alla modalità di gestione dei flussi vietati. Il bottone dedicato alla funzionalità dell'eliminazione di una strada è disabilitato se non ci sono strade da eliminare, ovvero se l'incrocio è vuoto.

Un controllo di coerenza viene effettuato anche per il bottone *Avanti>>*, che non permette di passare al prossimo step, se non sono verificate almeno queste condizioni:

a) L'incrocio deve avere almeno tre strade.
b) Se l'incrocio prevede una corsia entrante (o uscente) di una determinata tipologia, allora deve essere presente una corsia uscente (o entrante) della medesima tipologia, ma appartenente ad una strada differente. Questo controllo è dato dal fatto che si vuole evitare la creazione di incroci incoerenti, ovvero che prevedono una tipologia di traffico che può solamente entrare o uscire dall'incrocio.

In caso di errore dovuto al mancato rispetto della condizione *a*, il programma segnalerà l'errore attraverso una finestra come la seguente:

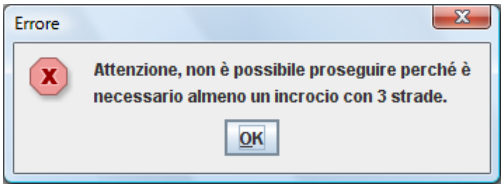


Figura 7 - Schermata di errore per incrocio con meno di 3 strade

Se invece l'errore è originato dal mancato rispetto della condizione *b*, il programma fornirà altresì una schermata di errore, ma cercherà anche di indicare quale tipologia di traffico ha causato l'incoerenza nell'incrocio. Un esempio di messaggio è il seguente:

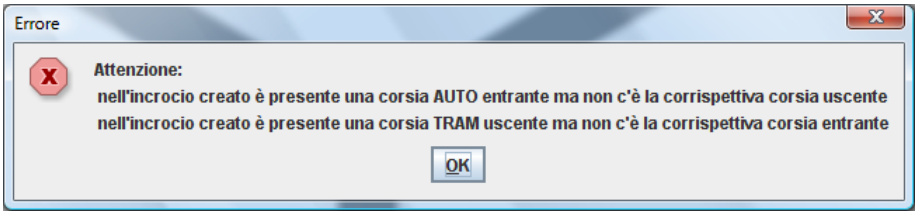


Figura 8 - Schermata di errore per incrocio non coerente

La pressione del bottone *Aggiungi Strada*, ci permette di aprire una nuova finestra, attraverso la quale è possibile aggiungere una nuova strada all'incrocio che stiamo disegnando. Per ogni strada è possibile specificare se possiede o meno l'attraversamento pedonale e se permette l'inversione. La checkbox che controlla l'inversione si abilita solamente quando la strada può effettivamente garantirla, ovvero quando ha almeno una corsia entrante ed una corsia uscente associate alla tipologia di traffico automobilistico. Inoltre, nell'interfaccia grafica della finestra, abbiamo due bottoni che ci permettono di inserire o eliminare corsie. Le corsie già inserite possono essere controllate grazie alla tabella di riepilogo, implementata con una *JTable* non editabile. Per ogni corsia è riportato l'indice e la tipologia di traffico associata.

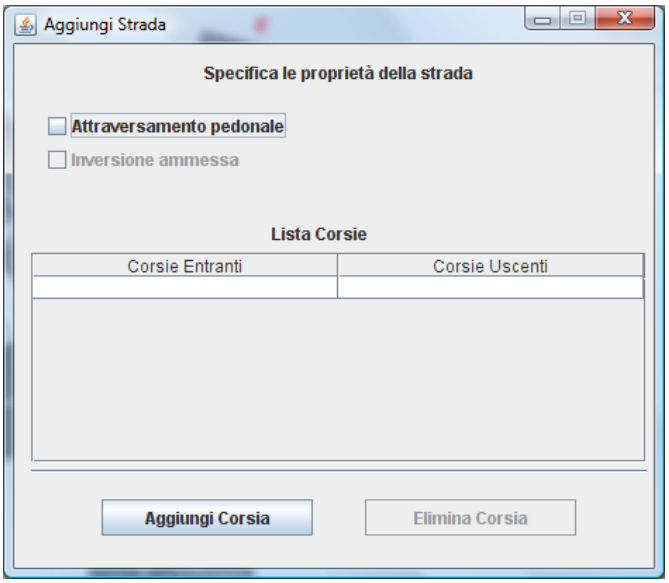


Figura 9 - Finestra per aggiungere una Strada

La pressione del pulsante *Aggiungi Corsia* apre una ulteriore finestra attraverso la quale è possibile specificare il tipo di corsia da aggiungere. La corsia può essere di tipo Entrante o di tipo Uscente, ed inoltre si può selezionare anche la tipologia di traffico associata, abilitando l'opportuno radio button su Auto o Tram. In caso di corsia Entrante sarà possibile scegliere solamente

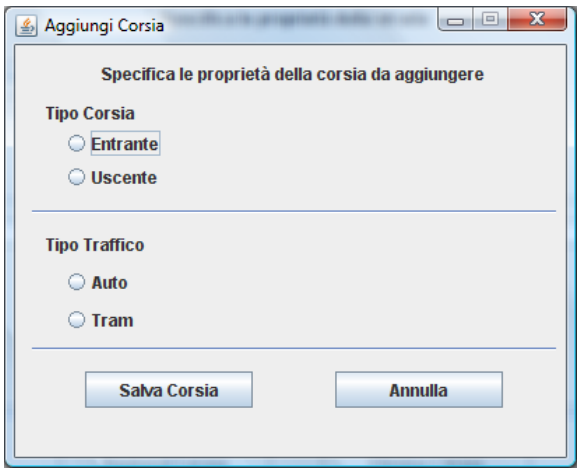


Figura 10 - Finestra per aggiungere una Corsia

uno dei valori tra Auto e Tram, mentre per le corsie uscenti questo vincolo non è presente e quindi si possono scegliere entrambi i valori. Nel caso in cui si andasse ad inserire una Corsia di tipo Entrante o Uscente, con la medesima tipologia di traffico di un'altra corsia della stessa strada (rispettando sempre il tipo della corsia), si verrebbe avvisati dell'errore commesso attraverso una schermata come la seguente:

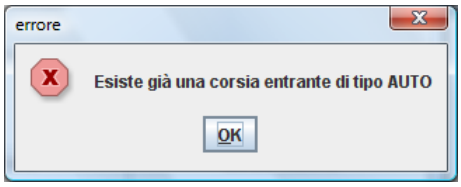


Figura 11 - Schermata di errore in caso di Corsia già specificata

Iterando la procedura di aggiunta di una strada e delle sue corsie, si può arrivare alla definizione della topologia dell'incrocio, come si può vedere da questa figura.

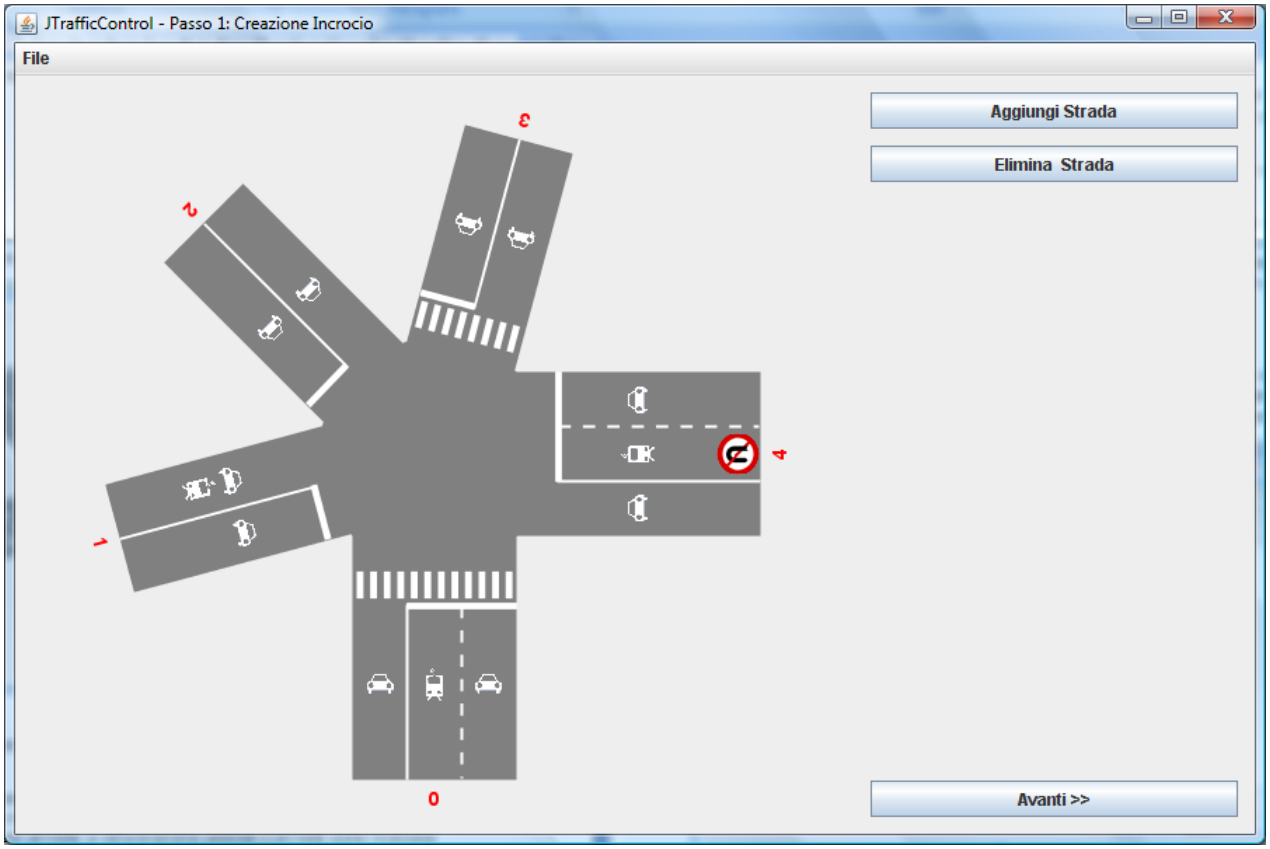


Figura 12 - Esempio di un incrocio disegnato da JTrafficControl

Si può notare infatti che questo incrocio è formato da 5 strade (numerate da 0 a 4) con le seguenti caratteristiche:

0) Questa strada ha 3 corsie, due entranti ed una uscente. Le due corsie entranti sono rispettivamente associate a traffico tramviario e automobilistico, mentre la corsia uscente è associata al solo traffico auto. La strada permette un attraversamento pedonale e permette anche l'inversione.
1) Questa strada è composta da due corsie, una entrante ed una uscente. La corsia entrante prevede solamente del traffico auto, mentre la corsia uscente è associata al traffico automobilistico e tramviario. La strada non ha attraversamenti pedonali ma permette l'inversione.

2) Questa strada ha 2 corsie, una entrante ed una uscente, entrambe associate a del traffico auto. Così come per la strada 1, la strada 2 non permette attraversamento pedonale ma permette l'inversione del senso di marcia.
3) Questa strada ha le stesse caratteristiche della precedente, alle quali va aggiunta quella dell'attraversamento pedonale.
4) La strada ha 3 corsie, due entranti ed una uscente, rispettivamente associate al traffico automobilistico, al traffico tramviario e nuovamente al traffico auto. La strada non permette attraversamento pedonale e nemmeno inversione del senso di marcia.

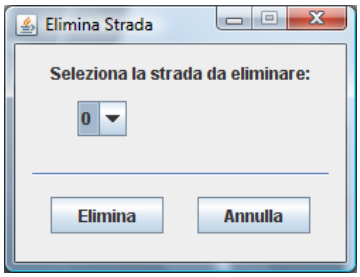


Figura 13 - Finestra per l'eliminazione di una Strada

Come si può notare dalla figura, adesso il bottone *Elimina Strada* è abilitato e premendolo si apre una finestra aggiuntiva che permette di selezionare la strada da eliminare.

La finestra ha un comportamento molto semplice e chiaro: attraverso la combo box è possibile selezionare la strada da eliminare, operazione che deve essere confermata attraverso la pressione del pulsante *Elimina*. Nel caso in cui non si fosse sicuri della volontà di eliminare una strada, è sempre possibile desistere dall'effettuare l'operazione premendo il bottone *Annulla*.

Ritornando al frame principale del programma, essendo l'incrocio coerente, premendo il tasto *Avanti>>* si passa al prossimo step di esecuzione che praticamente lancia l'algoritmo di numerazione e permette di definire i flussi vietati. La schermata che ci troviamo di fronte è la seguente:

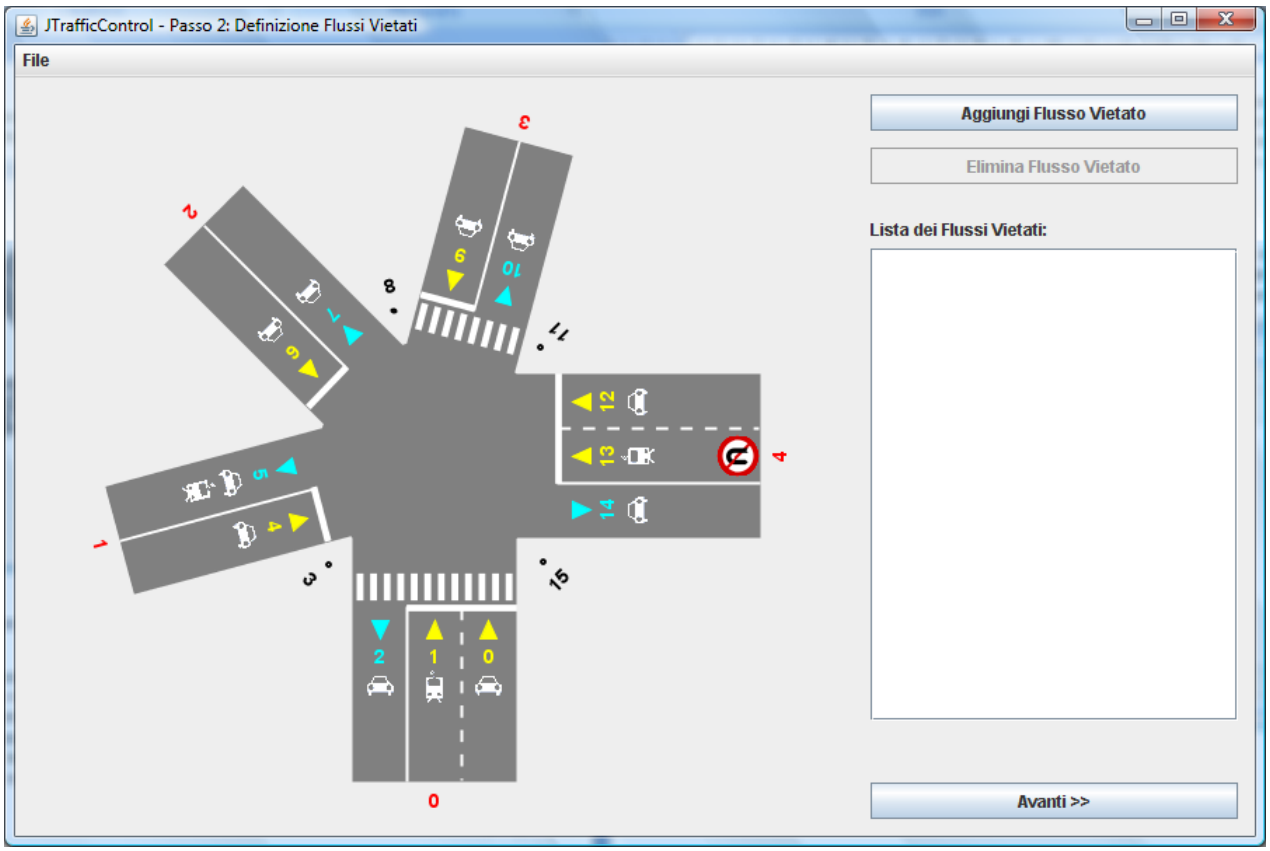


Figura 14 - Secondo passo di JTrafficControl

Come si può vedere dalla figura, adesso l'incrocio ha subito la numerazione di tutti i punti di ancoraggio dei flussi. Con il triangolo giallo vengono denotati i punti sorgente, con il triangolo celeste vengono evidenziati i punti di destinazione mentre con il punto nero viene evidenziato il punto di ancoraggio dell'attraversamento pedonale.

L'algoritmo procede con la numerazione dei punti attraverso le regole che abbiamo descritto precedentemente nel modello: si inizia dalla prima corsia entrante (se presente) della strada 0, iniziando l'assegnazione numerica a partire dal numero 0, procedendo in senso orario. Avendo a disposizione tutti i numeri delle sorgenti e destinazioni, è possibile procedere alla descrizione dei flussi vietati, che ovviamente non possono prendere in considerazione punti di ancoraggio per l'attraversamento pedonale. Per aggiungere un flusso vietato è sufficiente cliccare sul bottone *Aggiungi Flusso Vietato*, che apre una nuova finestra del tutto simile alla figura seguente.

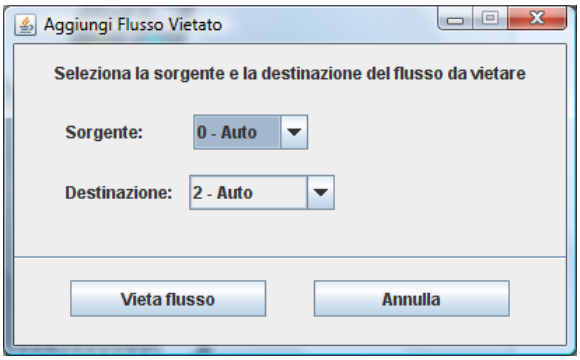


Figura 15 - Finestra per aggiungere un flusso vietato

Un flusso vietato deve essere specificato a partire dalla sua sorgente e dalla destinazione. Per facilitare l'inserimento di queste informazioni sono state predisposte due apposite combo box che riportano distintamente le sorgenti e la tipologia di traffico associata, nonché le destinazioni con sempre associata la tipologia di traffico. Una volta scelta la sorgente e la destinazione è possibile vietare il flusso premendo il bottone *Vieta flusso*; se invece non si è sicuri dell'operazione è sempre possibile annullarla premendo il bottone *Annulla*.

In caso di conferma della volontà di vietare il flusso, il programma esegue un controllo di coerenza, andando a verificare che la sorgente e la destinazione siano associate al medesimo tipo di traffico. Se così non fosse viene restituito un messaggio di errore come il seguente.

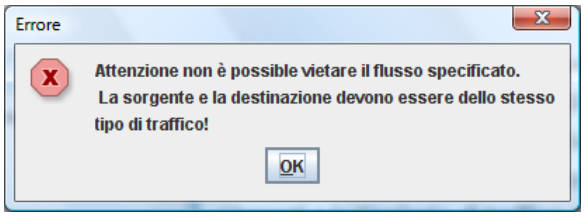


Figura 16 - Finestra di errore per un flusso vietato incoerente

In caso di esito positivo invece il flusso viene registrato come vietato e viene incluso nella textarea della schermata principale, abilitando di fatto anche il bottone *Elimina Flusso Vietato*.

La pressione del bottone *Elimina Flusso Vietato* apre una ulteriore finestra, attraverso la quale è possibile scegliere il flusso vietato che si vuole eliminare. Seguendo l'approccio già definito per le altre schermate di eliminazione, la lista dei flussi vietati viene presentata attraverso una combo box dove, una volta che si è selezionato l'elemento da eliminare, si può dare la conferma attraverso la pressione del pulsante *Elimina*. Il bottone *Annulla* permette di chiudere la finestra e di lasciare tutto inalterato.

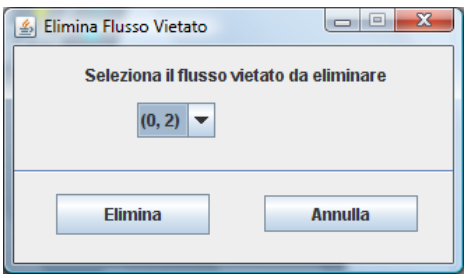


Figura 17 - Finestra per l'eliminazione di un flusso vietato

La pressione del bottone *Avanti>>*, in questo step di esecuzione, permette all'utente di proseguire nell'interazione e di approdare al terzo passo, dove vengono presi in considerazione i vincoli temporali da stabilire sui flussi dell'incrocio. Ci troviamo di fronte alla seguente schermata, dove a lato del disegno dell'incrocio, troviamo un pannello con alcune opzioni selezionabili.

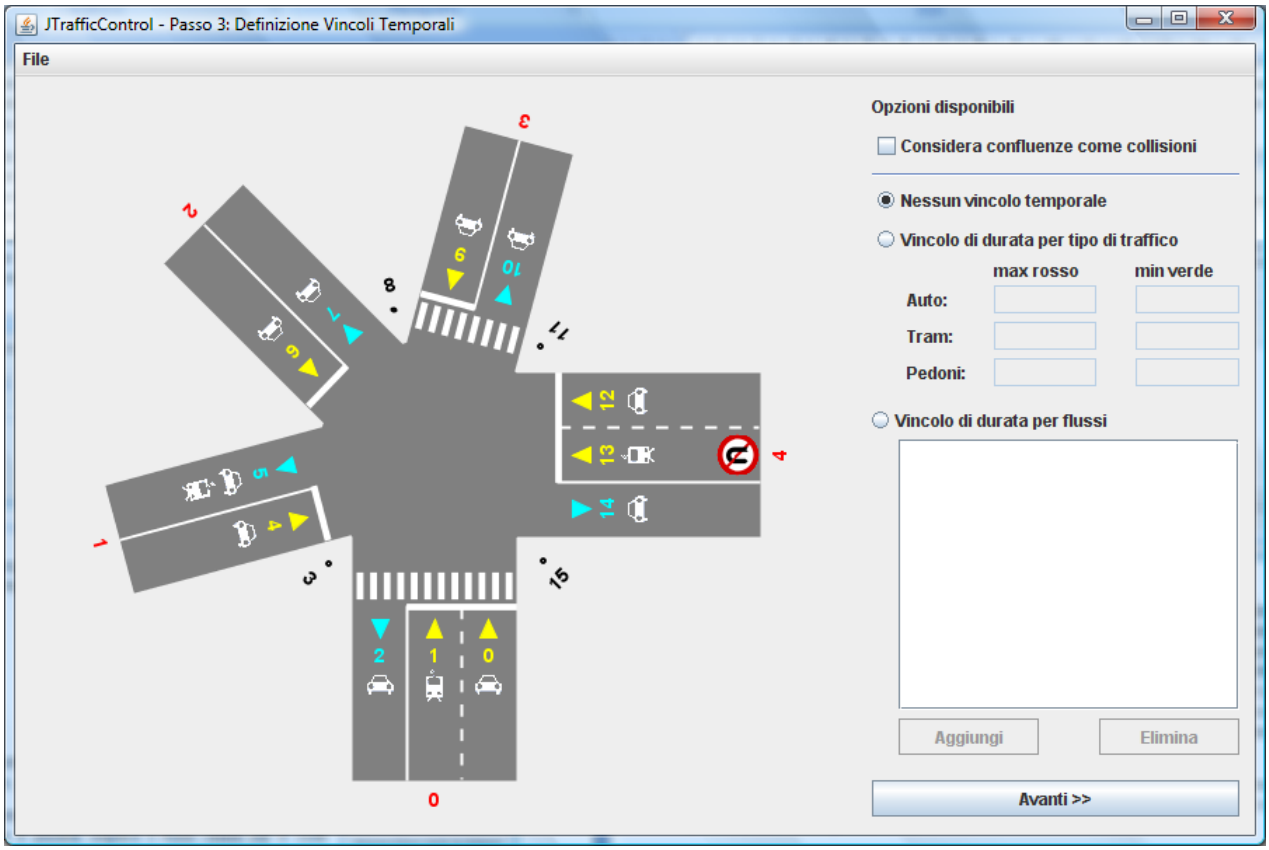


Figura 18 - Terzo passo di JTrafficControl

La prima opzione selezionabile, in alto a sinistra, è quella riguardante il trattamento delle confluenze, ovvero di quei flussi che hanno la medesima destinazione ovvero che confluiscono nella stessa corsia uscente. Il programma, di default, non considera le confluenze come collisioni, ma questa scelta può essere modificata selezionando la voce *Considera confluenze come collisioni*. Difatti se la scelta di considerare le confluenze come collisioni viene abilitata, il programma, in caso di esito positivo, ritornerà uno schedule che non permette di abilitare contemporaneamente il semaforo verde per nessuna coppia confluente dell'incrocio. Ovviamente questa scelta irrigidisce il sistema, ponendo su di esso un vincolo forte e stringente, che spesso fa sì, abbinandolo anche ai vincoli temporali, che lo schedule non venga trovato.

Di tutt'altra natura sono invece i vincoli immediatamente sotto a quello appena descritto: permettono di impostare, qualora fosse necessario, vincoli temporali sui flussi. Le strategie a disposizione sono tre e non sono selezionabili contemporaneamente:

- *Nessun vincolo temporale*: di default è l'opzione selezionata. Specifica al programma che non si vuole impostare nessuna forma di vincolo temporale per l'incrocio che si sta considerando.
- *Vincolo per tipo di traffico*: selezionando questa opzione invece è possibile specificare vincoli temporali per tipologia di traffico, ovvero per tutti quei flussi di tipo auto, tramviario e pedonali. I vincoli esprimibili sono sulla durata massima del rosso e sulla durata minima del verde, da considerare come istanti, ovvero turni dello schedule. Se si sceglie questa opzioni non si è costretti

a definire vincoli temporali per tutti i flussi, così come non si è costretti a specificare, per un tipo di flusso, sia la durata massima del rosso che quella minima del verde, potendo quindi specificare anche solamente uno tra questi due vincoli.

- *Vincolo per flussi*: selezionando questa opzione è possibile specificare i vincoli temporali, sempre sulla durata massima del rosso e quella minima del verde, su un preciso flusso. L'aggiunta dei vincoli è possibile premendo sul bottone *Aggiungi*, mentre l'eliminazione di un vincolo su un flusso è possibile premendo il bottone *Elimina*. I due bottoni sono abilitati solamente quando l'opzione del vincolo per flussi è selezionata.

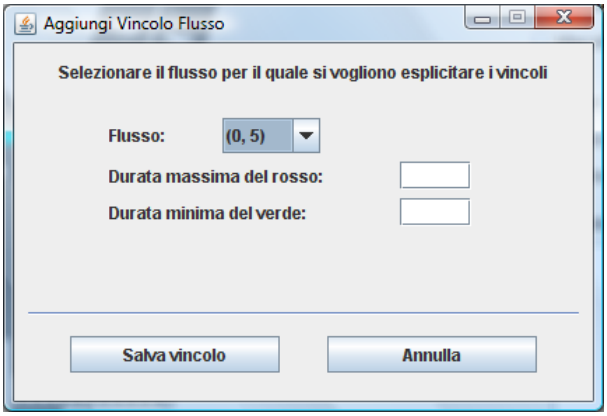


Figura 19 - Finestra per inserire vincoli temporali su uno specifico flusso

Nel caso in cui volessimo aggiungere un vincolo per flusso, dopo aver selezionato l'opportuno radio button, dobbiamo premere sul bottone *Aggiungi*, che di fatto apre questa nuova finestra. Attraverso la combo box è possibile selezionare il flusso per il quale vogliamo specificare i vincoli temporali, mentre grazie alle due textfield possiamo inserire i valori riguardanti la durata massima del rosso e la durata minima del verde. Premendo sul bottone *Salva vincolo*, il vincolo viene salvato, mentre premendo sul bottone *Annulla*, la finestra viene chiusa e non viene effettuata nessuna modifica.

Una volta che è stato specificato almeno un vincolo di durata per flussi, si abilita anche il bottone *Elimina*, che apre la finestra mostrata nella figura qui affianco. Attraverso la combo box è possibile selezionare quale vincolo sul flusso si vuole eliminare, per poi confermare l'operazione di rimozione premendo sul bottone *Elimina*. Nel caso in cui non si fosse certi, si può sempre premere il bottone *Annulla* per chiudere la finestra e ritornare al frame principale, senza apportare nessuna modifica.

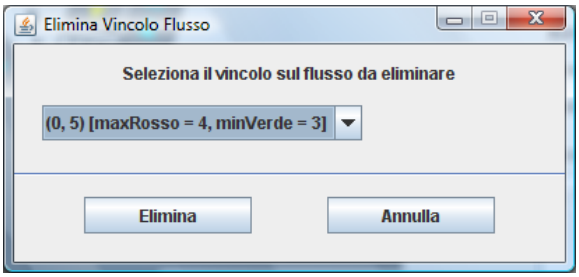


Figura 20 - Finestra per eliminare un vincolo su di un flusso specifico

Ritornando al frame principale, premendo il bottone *Avanti>>* si apre l'ultima schermata del programma, da dove è possibile lanciare l'esecuzione con NuSMV e controllare l'output generato.

L'ultima schermata è la seguente:

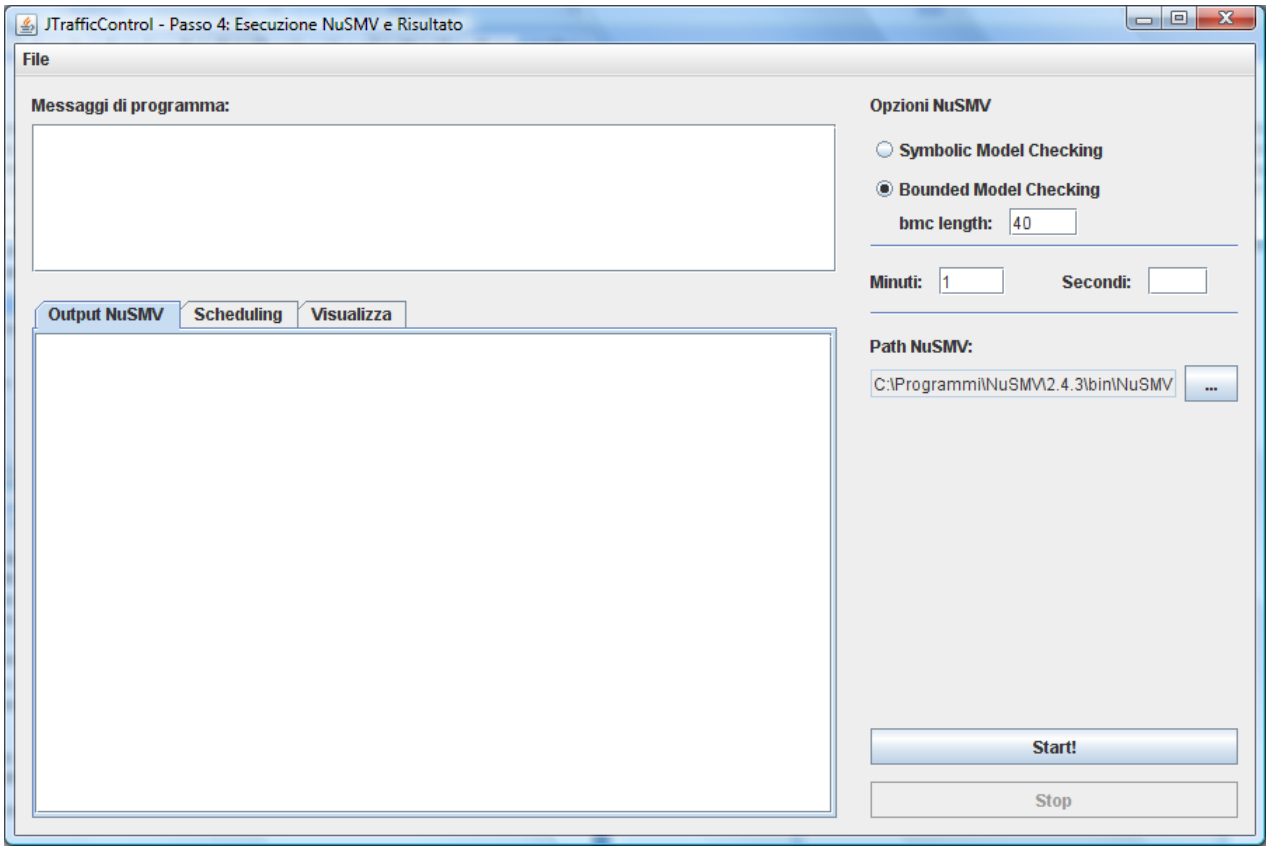


Figura 21 - Schermata finale di JTrafficControl

Il pannello di sinistra è suddiviso a sua volta in due pannelli: il primo raccoglie i messaggi generati dal programma Java durante l'esecuzione, mentre il secondo pannello non è altro che un *JTabbedPane* con 3 tab. Il primo tab, *Output NuSMV*, ripropone fedelmente la stringa di output generata da NuSMV; il secondo tab, *Scheduling*, propone una tabella con tutti i valori flusso per flusso istante per istante; il terzo ed ultimo tab, *Visualizza*, permette di controllare istante per istante quali sono i flussi attivi, visualizzandoli direttamente sul disegno dell'incrocio.

Il pannello di destra permette di gestire alcuni parametri di esecuzione di NuSMV e nello specifico si può scegliere se utilizzare il *Symbolic Model Checking*, piuttosto che il *Bounded Model Checking*; stabilire dopo quanti minuti e secondi scatta il timeout che blocca l'esecuzione ed infine scegliere il path dell'eseguibile di NuSMV. In fondo al pannello troviamo il bottone *Start!* ed il bottone Stop, che rispettivamente permettono di far partire e terminare la computazione.

Con l'esempio utilizzato, dopo la pressione del tasto *Start!* otteniamo le seguenti schermate di output:

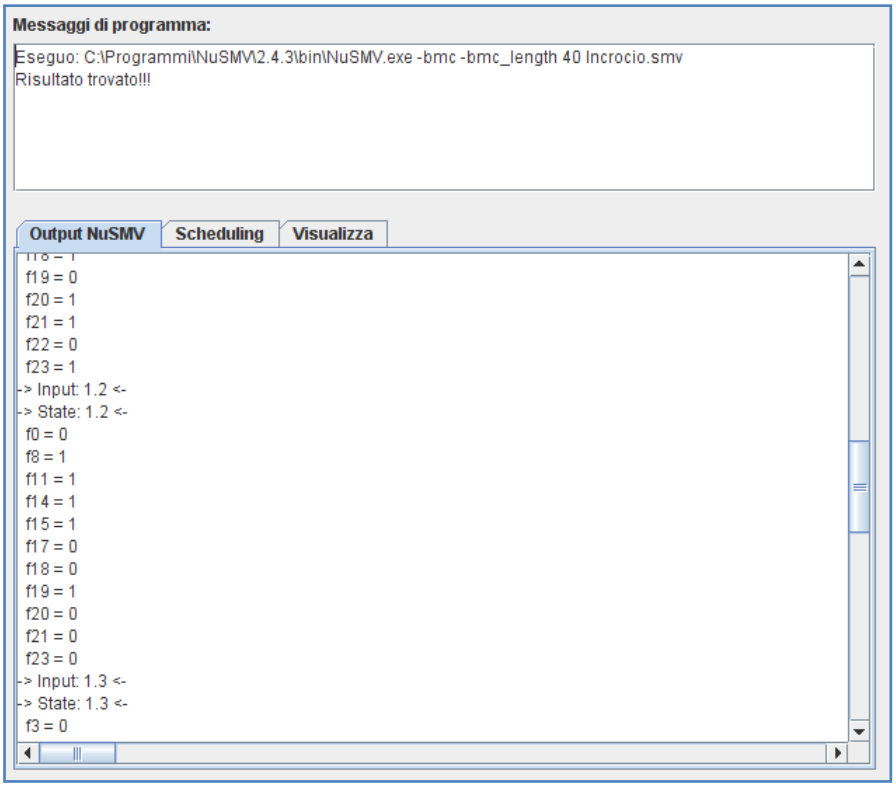


Figura 22 - Messaggi del programma e visualizzazione output NuSMV

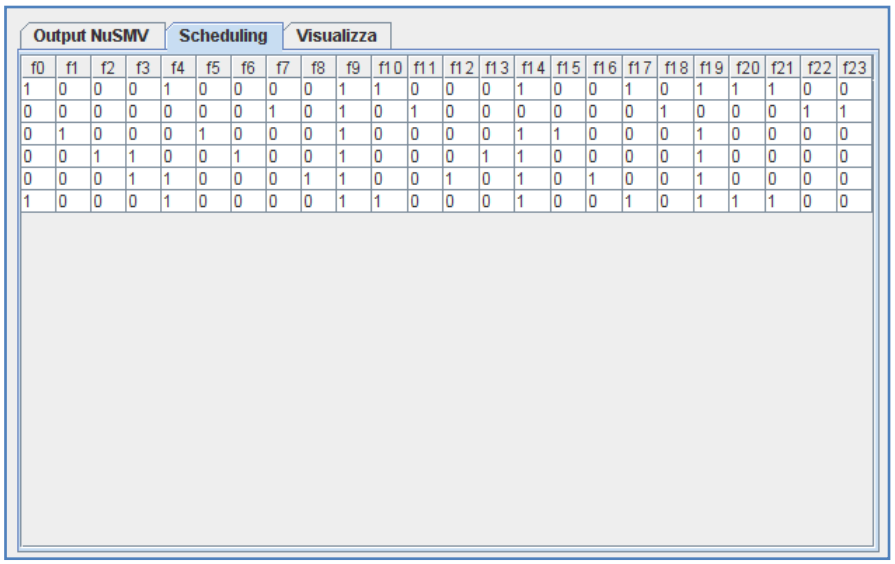


Figura 23 - JTable con i valori dello schedule ottenuto

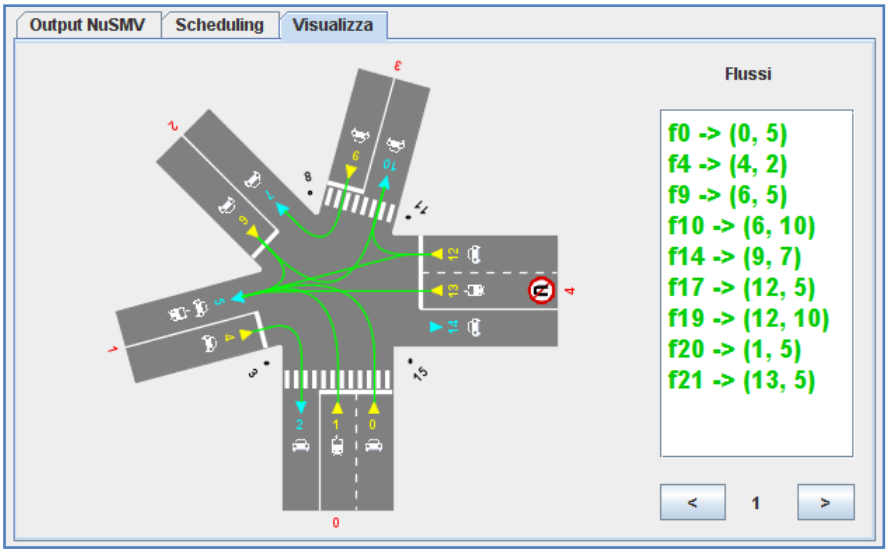


Figura 24 - Visualizzazione grafica dei turni dello schedule ottenuto

Esempi Pratici

In questa sezione della relazione presentiamo alcuni esempi di incrocio e valori di output che abbiamo raccolto, come prova del funzionamento del programma.

Primo esempio

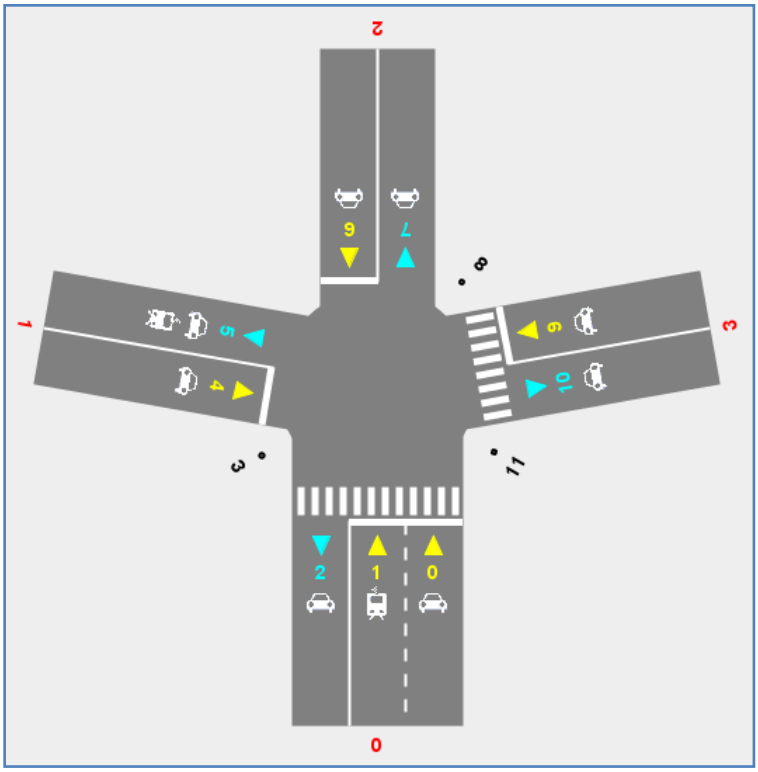


Figura 25 – Incrocio del primo esempio pratico

Il primo esempio pratico è basato sull'incrocio rappresentato nella figura qui a fianco. Si tratta di un incrocio molto semplice, composto da 4 strade, dove per ognuna è permessa l'inversione di marcia ma solamente 2 hanno anche un attraversamento pedonale.

Nell'incrocio è presente anche un flusso tramviario, che parte dalla strada 0, verso la strada 1.

In questo esempio pratico non vietiamo nessun flusso, accettiamo il comportamento di default sulle confluenze ma andremo ad indicare vincoli temporali sui flussi, in base alla loro tipologia.

Nello specifico, vogliamo uno schedule che soddisfi i seguenti vincoli temporali:

- Flusso auto: per questi tipi di flussi impostiamo la durata massima del rosso pari a 6 istanti, mentre la durata minima del verde la settiamo pari a 2.
- Flusso tramviario: per questi tipi di flusso usiamo le stesse impostazioni del flusso auto, ovvero una durata massima del rosso pari a 6 istanti e la durata minima del verde pari a 2.
- Flussi pedonali: per i flussi pedonali impostiamo una durata massima del rosso pari a 6 istanti, ed una durata minima del verde pari a 4 istanti.

Con questa topologia di incrocio e con questi vincoli temporali, cercando la soluzione con NuSMV ed impostando una ricerca di tipo Bounded Model Checking, con limite massimo di lunghezza pari a 40, otteniamo un risultato interessante: non esiste schedule in grado di soddisfare i vincoli così come sono stati impostati. Il programma allora esegue l'algoritmo che spezza il flusso con il maggior numero di collisioni e solo così riesce a trovare una soluzione.

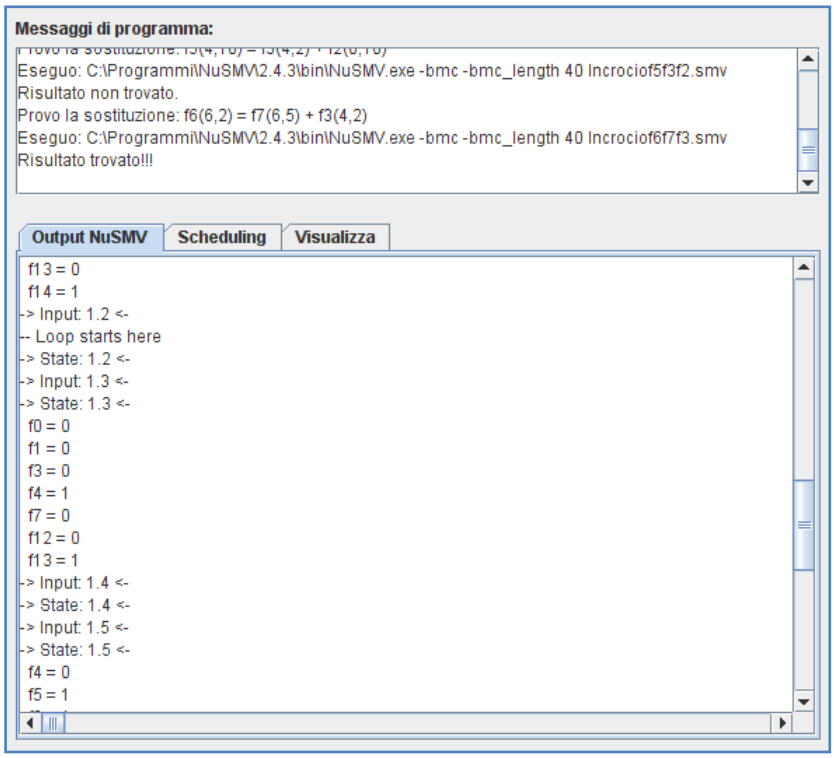
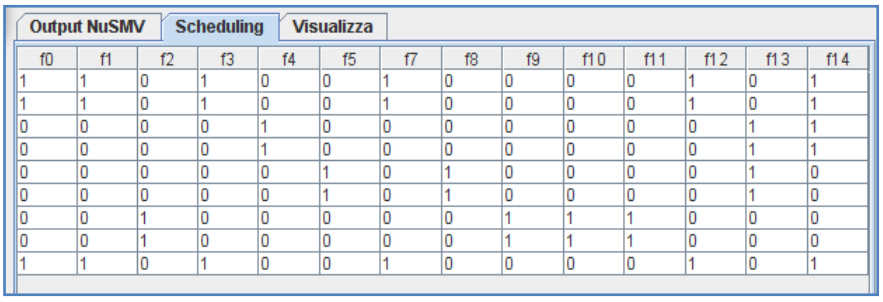


Figura 26 - Finestra con il risultato della computazione

Come si può vedere dalla figura, il risultato viene trovato alla quinta prova, con il file *Incrociof6f7f3*.smv, ovvero con il flusso f6 che viene spezzato nei flussi f7 + f3.

Lo scheduling dei flussi può essere visualizzato nella sua interezza tramite il tab *Scheduling*, che in questo caso presenta a video i risultati così come sono visualizzati nella figura qui riproposta. Ogni colonna rappresenta un flusso, mentre ogni riga rappresenta un istante, o turno, dello schedule trovato. Quando il valore del flusso è pari a 1 significa che quel flusso è abilitato, ovvero che ha il semaforo con valore verde; invece quando il valore del flusso è pari a 0 significa che in quell'istante gli elementi del flusso non possono circolare e quindi il semaforo è rosso.



f0	f1	f2	f3	f4	f5	f7	f8	f9	f10	f11	f12	f13	f14
1	1	0	1	0	0	1	0	0	0	0	1	0	1
1	1	0	1	0	0	1	0	0	0	0	1	0	1
0	0	0	0	1	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	1	0	0	0	0	1	0
0	0	0	0	0	1	0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	1	1	0	0	0
0	0	1	0	0	0	0	0	1	1	1	0	0	0
1	1	0	1	0	0	1	0	0	0	0	1	0	1

Figura 28 - Valori dello schedule ottenuto

Tramite il tab *Visualizza* possiamo invece osservare visivamente i flussi abilitati, e procedere con i bottoni **>** e **<** per passare da un istante all'altro.

Durante il primo istante sono abilitati i flussi f0, f1, f3, f7, f12 e f14. Come si può vedere dalla figura il flusso f14 non rappresenta altro che l'attraversamento pedonale della strada 3; d'altra parte i flussi f7, f12 e f0, invece, sono flussi confluenti e siccome non era stato imposto di considerare le confluenze come collisioni, vengono giustamente abilitati contemporaneamente.

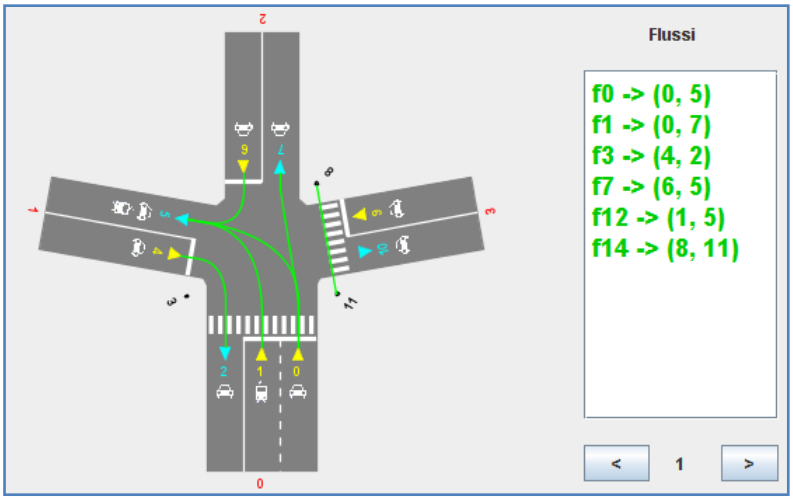


Figura 29 - Primo turno

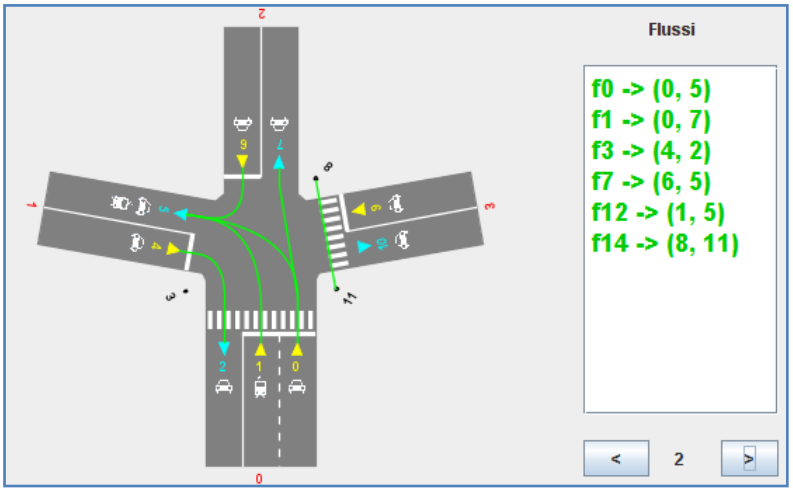


Figura 30 - Secondo turno

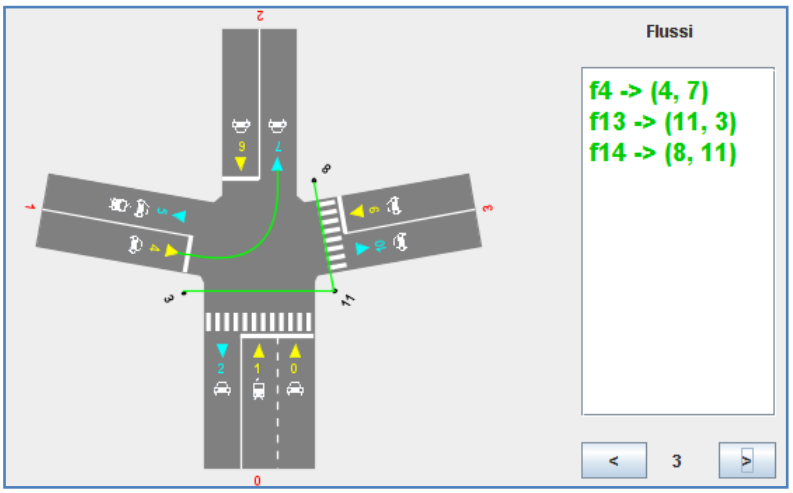


Figura 31 - Terzo turno

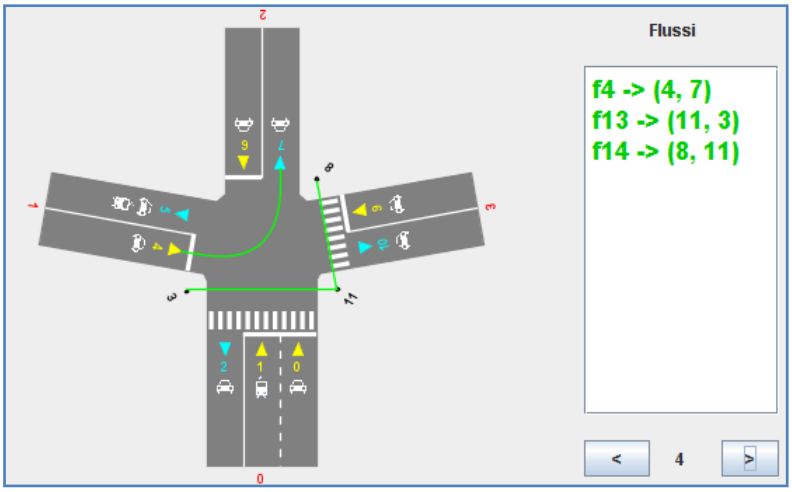


Figura 32 - Quarto turno

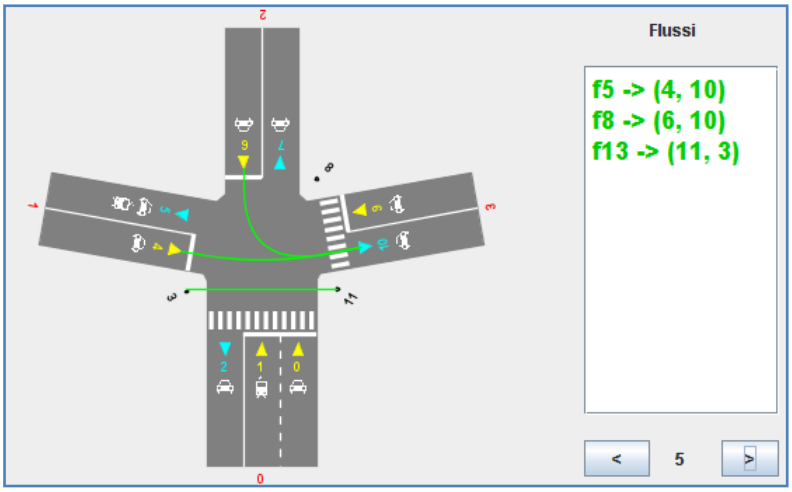


Figura 33 - Quinto turno

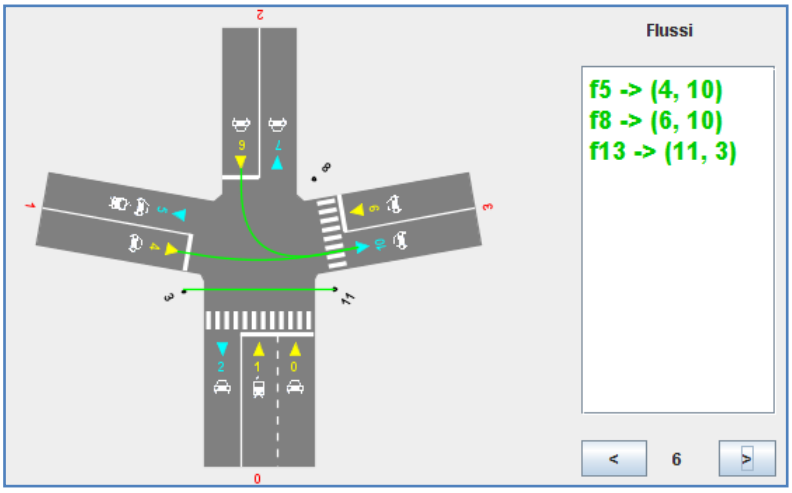


Figura 34 - Sesto turno

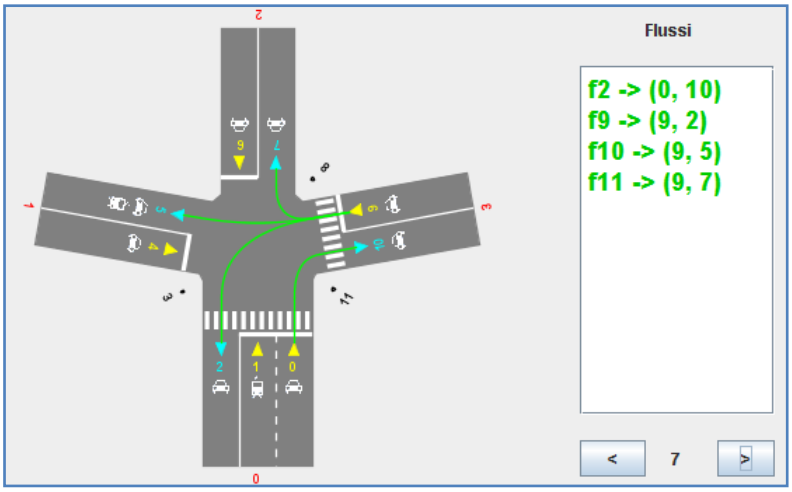


Figura 35 - Settimo turno

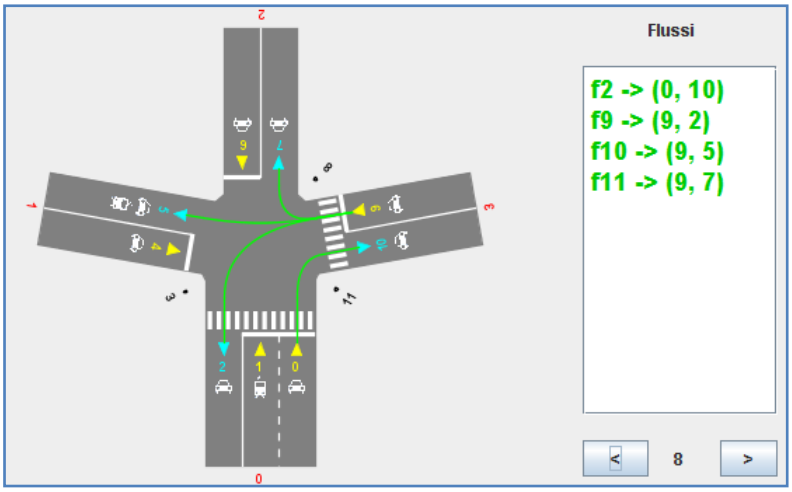


Figura 36 - Ottavo turno

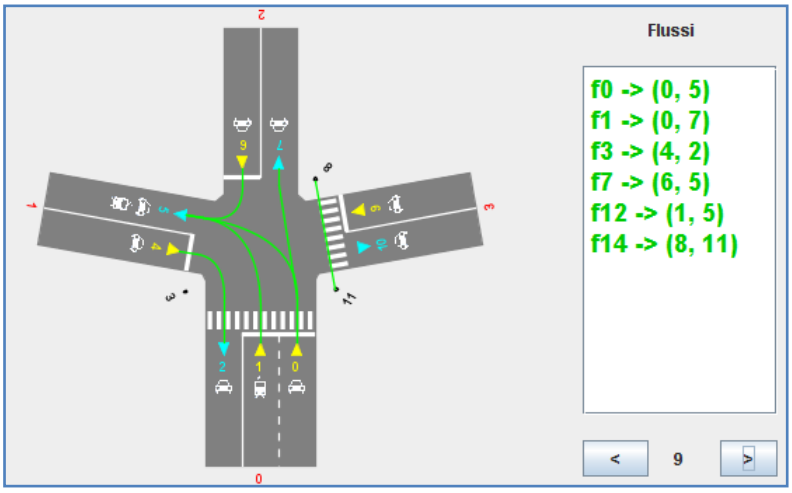


Figura 37 - Nono ed ultimo turno

Come si può notare dagli screenshot, i vincoli temporali che abbiamo imposto nell'esempio pratico sono rispettati: per esempio gli attraversamenti pedonali sono abilitati almeno per 4 istanti di tempo successivi, che corrispondono al vincolo della durata minima del verde, che precedentemente avevamo imposto pari a 4. Ugualmente si può notare che i flussi auto e tramviari sono abilitati per almeno 2 turni, dato che avevamo imposto il vincolo della durata minima del verde pari a 2.

Osserviamo anche che il flusso f6 non è presente nello schedule, perché sostituito dal flusso f7 + f3, che è abilitato sempre in coppia, ovvero quando è abilitato f7, lo è anche f3.

Secondo esempio

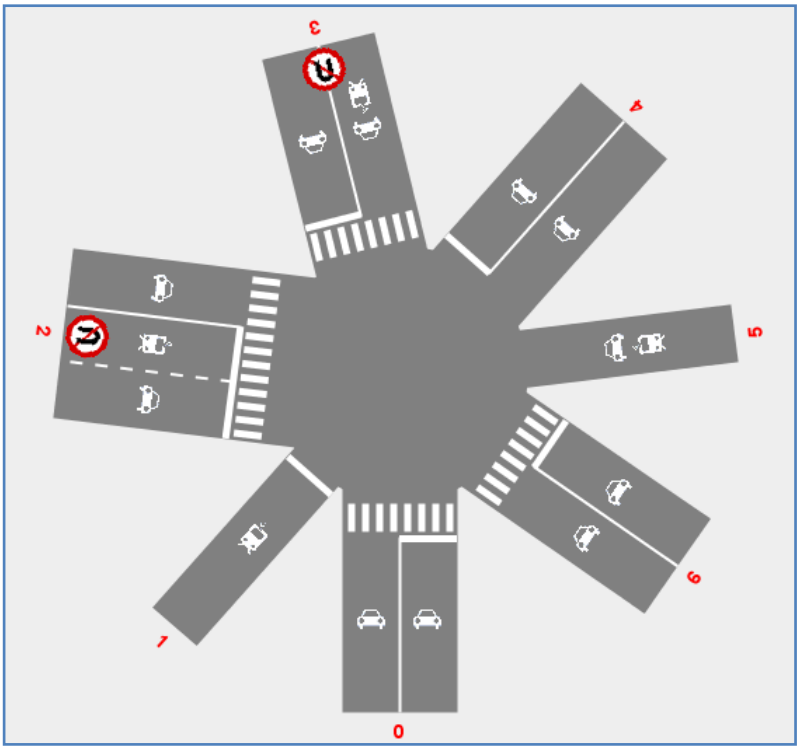


Figura 38 - Incrocio secondo esempio pratico

Il secondo esempio che prendiamo in esame è quello riportato nella figura qui affianco, dove sono presenti ben 7 strade, alcune con attraversamento pedonale, altre senza; alcune a senso unico ed altre a più corsie ma con il divieto di inversione di marcia.

Anche in questo caso numerico accetteremo il comportamento di default del programma, che non considera i flussi confluenti come collisioni.

Per quanto riguarda i vincoli temporali, impostiamo dei valori numerici per tipologia di traffico, seguendo queste indicazioni:

- Flusso auto: la durata massima del rosso è impostata a 10 istanti, mentre la durata minima del verde è impostata a 2.
- Flusso tramviario: la durata massima del rosso è settata ad 8 istanti, la durata minima del verde è pari a 1.
- Flusso pedonale: in questo caso la durata massima del rosso viene impostata a 8, mentre la durata minima del verde è pari a 3.

Data la complessità dell'incrocio e data la completezza dei vincoli temporali, che abbracciano tutti i flussi da considerare, abbiamo fatto girare il programma impostando il Bounded Model Checking, scegliendo come valore limite 20, pena la lunga attesa nella computazione.

Così come è impostato lo schedule non viene trovato con il modello di partenza, tantomeno con lo spezzamento di un solo flusso: affinché si arrivi ad una soluzione è stato necessario spezzare due flussi ed attendere la giusta combinazione. Il risultato finale si ottiene con il file *Incrociof14f10f4f21f23f16.smv*, ovvero con il flusso f14 suddiviso nei flussi f10 ed f4 e con il flusso f21 suddiviso anch'esso in due flussi, esattamente f23 ed f16.

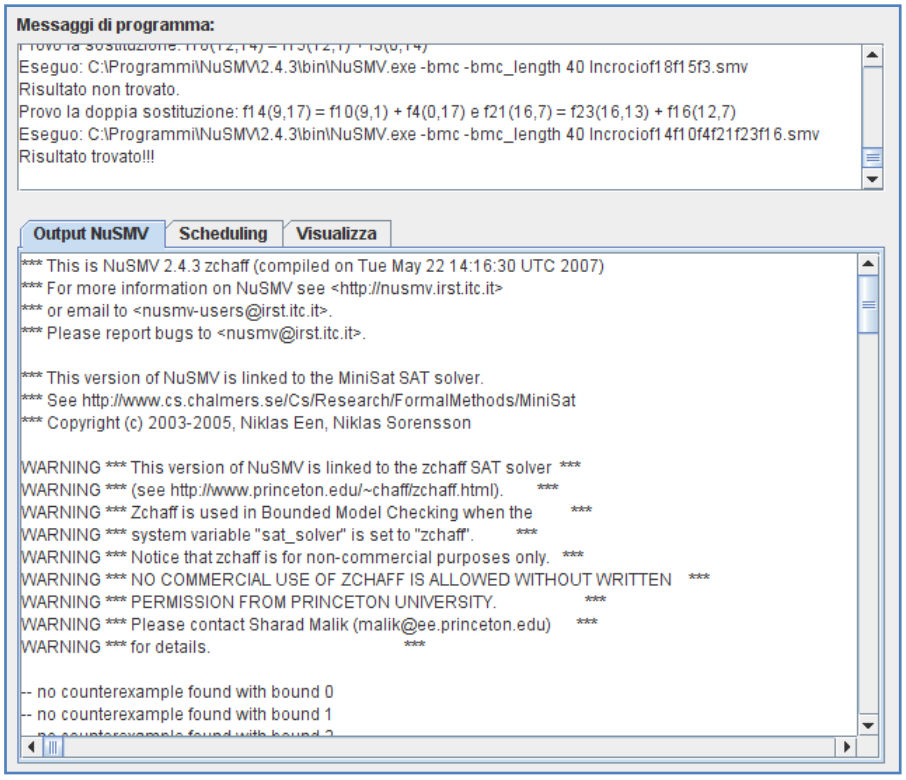
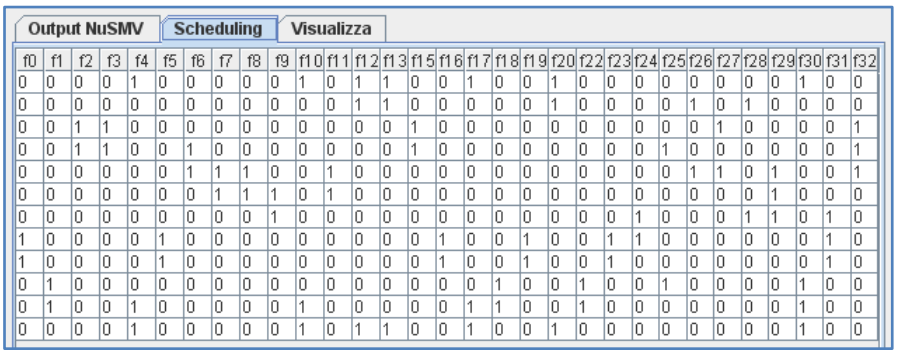


Figura 39 – Finestra con il risultato della computazione

Lo schedule trovato è molto complesso, e questo si può evincere guardando banalmente la tabella fornita in output.



f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f15	f16	f17	f18	f19	f20	f22	f23	f24	f25	f26	f27	f28	f29	f30	f31	f32
0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1
0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0

Figura 40 - Valori dello schedule ottenuto

Visualizzando i risultati dello schedule ottenuto, abbiamo i seguenti flussi, istante per istante:

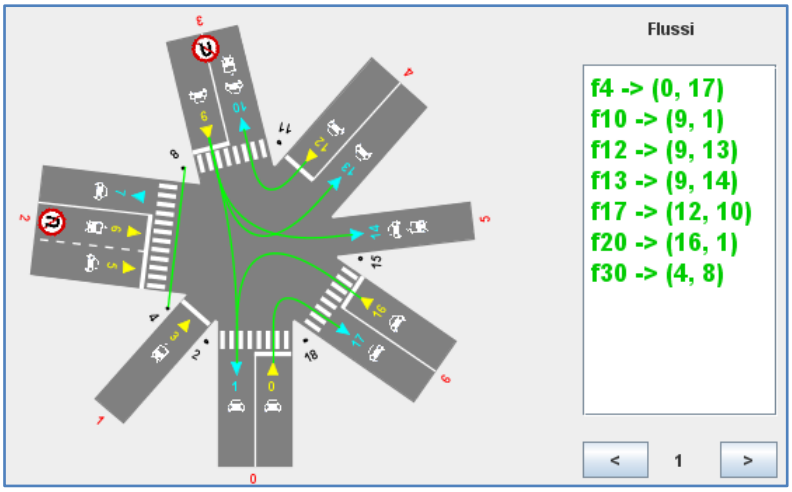


Figura 41 - Primo turno

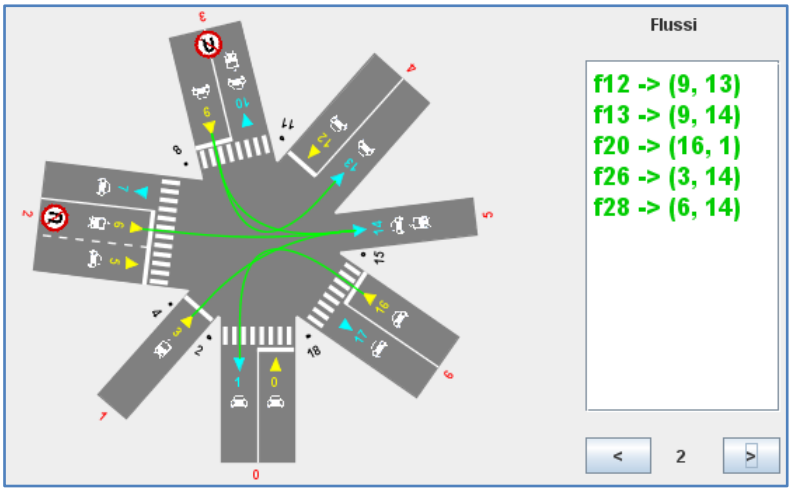


Figura 42 - Secondo turno

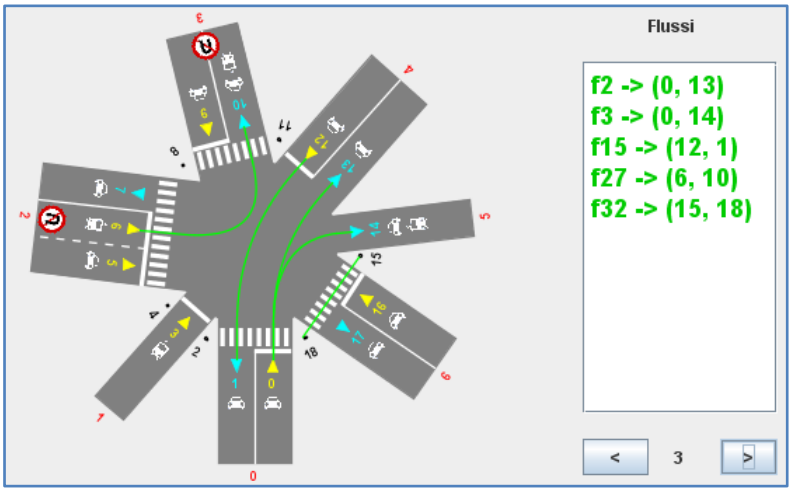


Figura 43 - Terzo turno

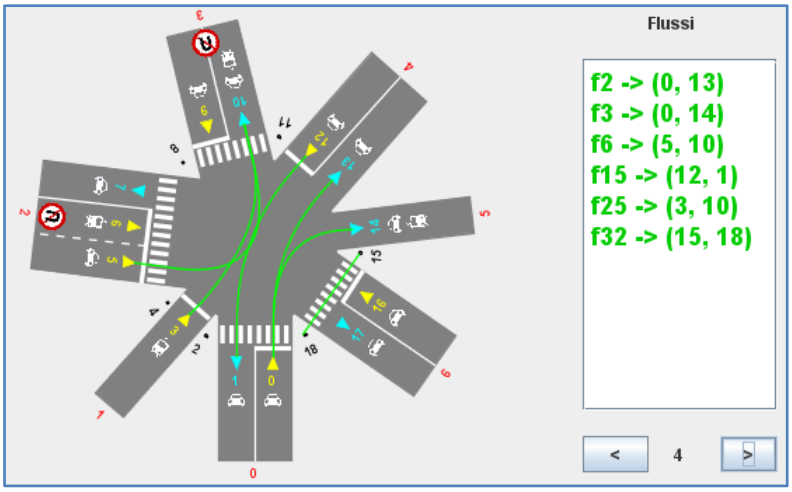


Figura 44 - Quarto turno

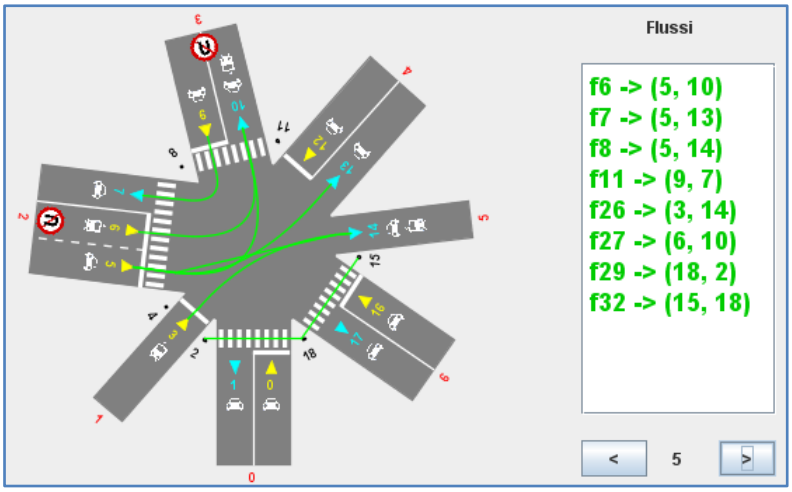


Figura 45 - Quinto tempo

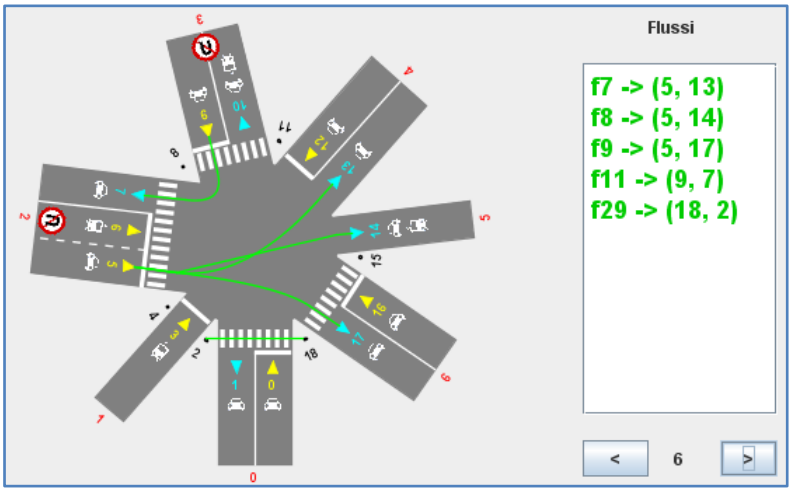


Figura 46 - Sesto turno

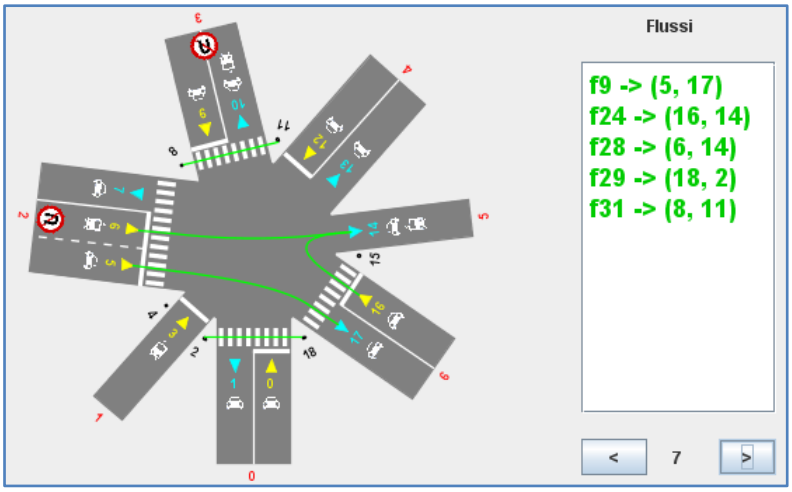


Figura 47 - Settimo turno

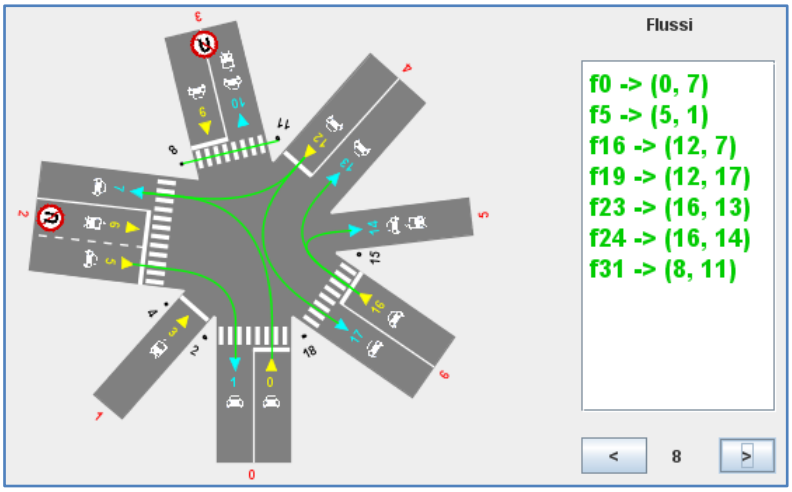


Figura 48 - Ottavo turno

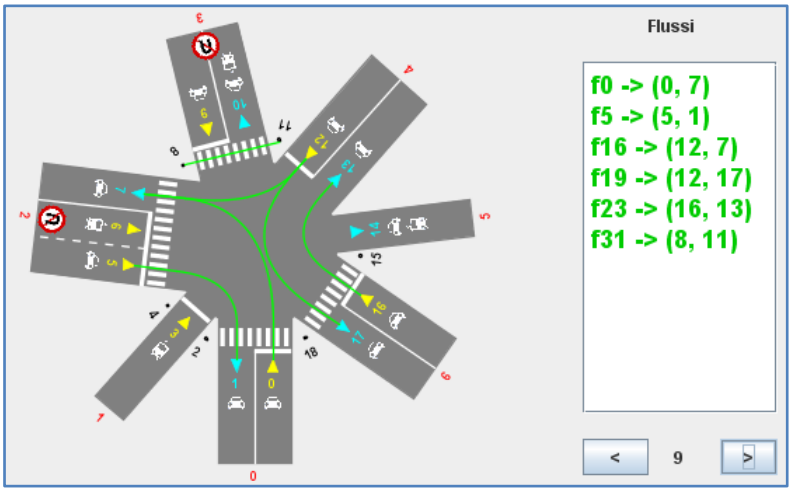


Figura 49 - Nono turno

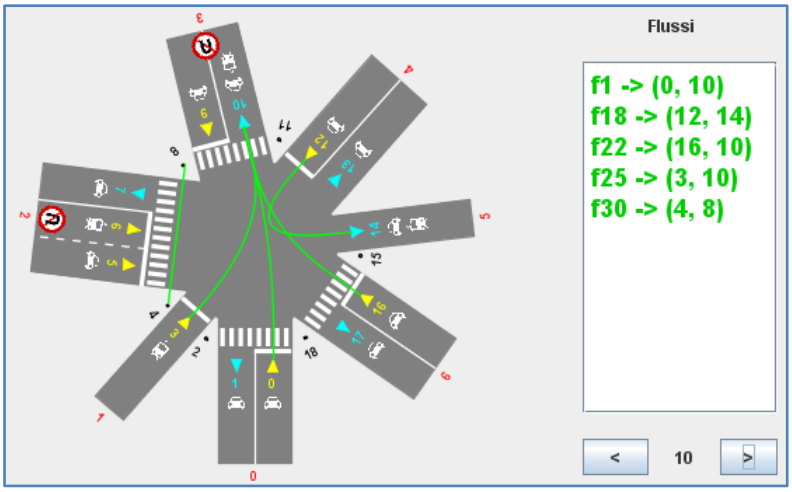


Figura 50 - Decimo turno

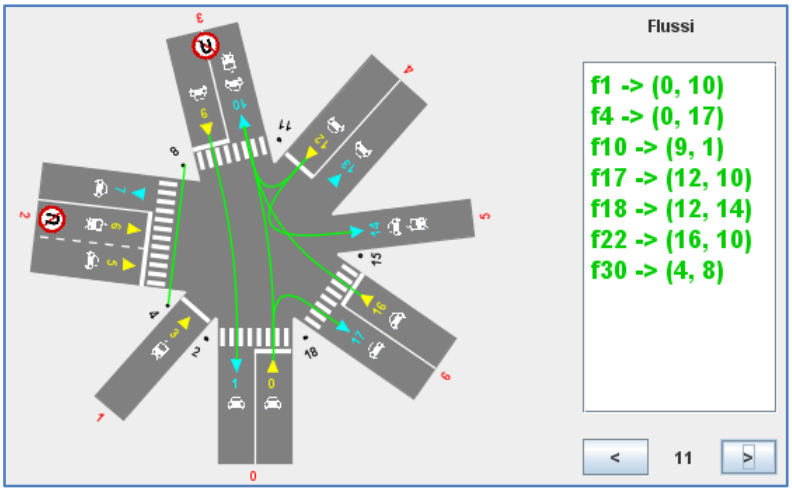


Figura 51 - Undicesimo turno

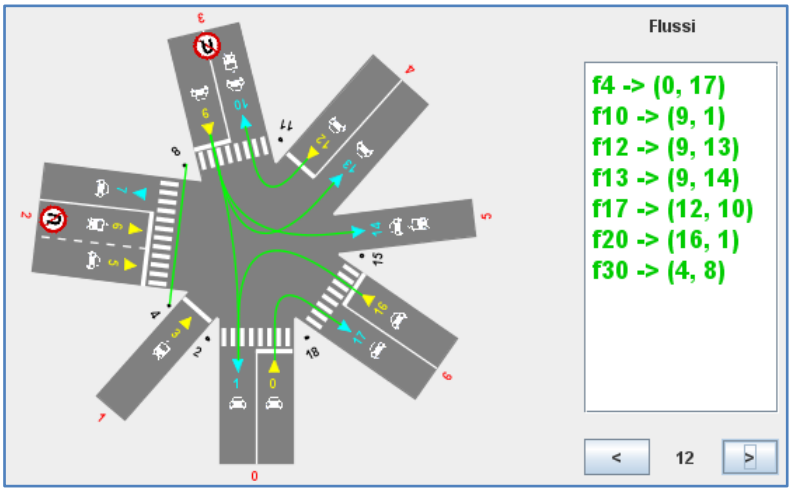


Figura 52 - Dodicesimo turno

Come si evince dal numero di turni, lo schedule trovato è sicuramente di difficile risoluzione in modo manuale: d'altronde si tratta di un incrocio con sette strade e con vincoli temporali su ogni flusso. Bisogna anche ricordarsi però che il risultato è frutto dello spezzamento di due flussi, altrimenti non si sarebbe giunti ad una soluzione.

Questo lavoro dimostra che il campo dei metodi formali, così come studiato nel corso, ben si presta alla verifica di determinate proprietà logiche (temporali e non), proprietà che possono essere facilmente contestualizzate in esempi concreti, come lo scheduling di un incrocio semaforico.