Model Checking Statistico per l'Opposite Direction in Vehicular Ad Hoc Network (VANET)

Prof. Ing. Floriano Scioscia
Mauro Losciale
Pietro Tedeschi

Linguaggi Formali e Compilatori
Laurea Magistrale in Ingegneria Informatica
Politecnico di Bari
A.A 2014 - 2015

Indice

1 Introduzione al progetto **3**
1.1 Introduzione ad Uppaal e al Model Checking Statistico . . . 3
1.1.1 Automi Temporizzati . . . 3
1.1.2 Caratteristiche fondamentali di Uppaal . . . 3
1.1.3 Il Model Checking Statistico . . . 5
1.1.4 Introduzione alle VANET . . . 6

2 Realizzazione del modello **8**
2.1 Analisi e descrizione generale del modello . . . 9
2.1.1 Rappresentazione dello scenario . . . 11
2.1.2 Parametri del modello di trasmissione . . . 13
2.1.3 Parametri del veicolo . . . 13
2.2 Modello del veicolo . . . 14
2.2.1 Descrizione dell'automa . . . 14
2.2.2 Descrizione del codice . . . 16
2.3 Modello dell'RSU . . . 21
2.3.1 Descrizione dell'automa . . . 21
2.3.2 Descrizione del codice . . . 21

3 Conclusioni e Risultati **22**

1 Introduzione al progetto

L'obiettivo del progetto, è stato quello di verificare mediante il Model Checking Statistico il modello di disseminazione Opposite direction nelle *Vehicular Ad Hoc Networks* (VANETs).

Nel dettaglio è stata verificata la percentuale dei veicoli che ricevono una segnalazione di un'anomalia di un punto $P(x, y)$ della mappa, quando questi si trovano ad una distanza maggiore o uguale a d da $P(x, y)$. I risultati sono stati verificati al variare della percentuale dei veicoli rispetto ad n (numero dei veicoli) e della distanza d.

La modellazione è stata effettuata mediante il model checker UPPAAL con il supporto **SMC** (Statistical Model Checking), al fine di consentire un'analisi stocastica efficiente delle proprietà relative alle performance degli automi temporizzati.

1.1 Introduzione ad UPPAAL e al Model Checking Statistico

UPPAAL è un Model Checker sviluppato dalle università di Uppsala (Svezia) ed Aalborg (Danimarca), e consiste in una serie di tool-box per la modellazione, simulazione e verifica di proprietà per i sistemi real time.

Esso permette di modellare tali sistemi attraverso reti di automi a stati finiti temporizzati, estesi tramite l'aggiunta di variabili intere, strutture dati, funzioni definibili dall'utente e canali di sincronizzazione. L'interfaccia utente è stata sviluppata utilizzando il linguaggio Java, mentre il tool per la verifica in C++ [6] [11] [1].

1.1.1 Automi Temporizzati

Il Model Checker UPPAAL è basato sulla teoria degli automi temporizzati. Un'automa temporizzato è definito come una macchina a stati finiti, il cui stato evolve scandito da un clock. E' possibile graficamente definire le diverse strutture di tali automi attraverso i **template**. Ogni istanza di tali template rappresenta un processo non deterministico durante l'evoluzione del sistema. I diversi processi (automi) interagiscono tra loro tramite **canali di sincronizzazione**.

La progressione dei clock avviene in maniera simultanea per tutti gli automi, e lo stato complessivo del sistema è definito considerando, per tutti gli automi, la *locazione corrente*, il *valore del clock* e delle *variabili globali* e *locali*. Ogni automa può percorrere un certo cammino chiamato **edge**, in maniera individuale, o sincrona rispetto ad altri automi presenti nel sistema, il quale permette la transizione dalla locazione corrente ad una nuova locazione, alterando lo stato del sistema.

Formalmente un automa temporizzato è definito da una tupla *(L, l_0, C, A, E, I)*, dove L è il set di locazioni, l_o è la locazione iniziale, C è il set dei clock, A è l'insieme delle azioni e co-azioni interne ed esterne, $E \subseteq L \times A \times B(C) \times 2C \times L$ è il set di edge e le relative azioni ammesse tra gli automi delle guardie e dei clock da resettare, ed infine $I : L \to B(C)$ è una funzione che assegna gli invarianti alle locazioni [6] [9] [11] [8].

1.1.2 Caratteristiche fondamentali di UPPAAL

Il linguaggio di modellazione di UPPAAL estende il concetto di automa temporizzato con le seguenti caratteristiche:

Templates: Gli automi potrebbero essere definiti mediante dei parametri di vario tipo (ad esempio, **int**, **chan**). I valori di tali parametri sono assegnati attraverso la definizione del processo nella sezione *Process Declarations*.

Costanti: Sono dichiarate tramite l'identificativo *const*. Per definizione non possono essere modificate ed il loro valore deve essere di tipo intero.

Variabili intere limitate: Sono dichiarate attraverso l'espressione *int[min, max]*, dove **min** e **max** rappresentano il valore minimo e massimo che la variabile può assumere rispettivamente. Guardie, invarianti ed assegnazioni possono contenere espressioni che coprono il range di valori di tali variabili. Inoltre viene effettuato un checking di tale range prima della verifica, e la violazione di tale range porta ad uno stato non valido che viene scartato a run-time.

Canali di sincronizzazione binaria: Sono dichiarati come *chan c*. Un edge che contiene un'etichetta **c!** si sincronizza con un altro edge contenente un'etichetta **c?**. Se sono ammesse più sincronizzazioni, la scelta viene effettuata in modo non deterministico.

Canali Broadcast: Nelle sincronizzazioni di tipo broadcast un mittente **c!** può sincronizzarsi con un numero arbitrario di riceventi **c?**. Ogni ricevente il cui stato corrente ammetta la sincronizzazione deve necessariamente eseguirla. Se non ci sono riceventi, il mittente esegue indipendetemente l'azione associata al comando **c!**, garantendo che la trasmissione in broadcast non venga bloccata.

Canali di sincronizzazione di tipo Urgent: Sono dichiarati tramite la keyword *urgent* come prefisso nella dichiarazione del canale. La transizione lungo una sincronizzazione urgent avviene senza ritardo. In questo caso gli edge non possono avere condizioni restrittive sui clock.

Locazioni di tipo Urgent: Quando il sistema si trova in uno stato di tipo **urgent** il tempo non scorre. Semanticamente tale proprietà ha la stessa valenza dell'aggiungere un clock x nel template, resettato a sua volta da tutti gli edge in ingresso allo stato considerato (si specifica nella sezione Update dell'edge l'istruzione $x = 0$), settando infine l'invariante $x <= 0$ nella locazione.

Locazioni di tipo Committed: Variante più restrittiva del tipo urgent. Il tempo non scorre e la transizione successiva tra tutti i processi del sistema dovrà essere l'uscita del processo dallo stato impegnato. E' utile per modellare operazioni atomiche e sincronizzazione tra processi.

Arrays: E' possibile dichiarare array di clock, canali, costanti e variabili intere. Tra parentesi quadre è specificata la dimensione dell'array. Alcuni esempi sono: **chan c[4], clock a[2], int[3,5] u.**

Inizializzatori: Sono utilizzati per inizializzare variabili intere, interi limitati ed array. Alcuni esempi sono: **int i = 2; or int i[3] = 1, 2, 3;.**

Tipi di dato Custom: Sono definiti tramite la sintassi C-like *typedef struct*. E' possibile definire qualsiasi tipo di dato estendendo i tipi base.

Funzioni utente: Possono essere definite sia a livello globale che a livello locale nei template. I parametri sono accessibili dalle funzioni definite localmente al template. La sintassi è molto simile al C, ma non prevede l'uso di puntatori.

Select: Una label di tipo Select contiene una o più espressioni del tipo *ID:Type*, separate da virgola, dove *ID* rappresenta il nome della variabile e *Type* il tipo associato. A seconda

del tipo associato, la variabile di nome *ID* può assumere tutti i valori possibili nel range dichiarato; verrà creato un edge per ognuno di questi valori. La variabile è accessibile solo lungo la transizione stessa, e la scelta del valore con cui effettuare tale transizione è non-deterministica.

Guardia: Una guardia è una particolare espressione che soddisfa le seguenti condizioni: è *side-effect free* (ovvero non altera il contenuto delle variabili); ha un valore di ritorno di tipo booleano; sono ammessi parametri di tipo clock, interi e costanti; clock e differenze di clock possono essere confrontati con espressioni di tipo intero; le guardie sui clock sono essenzialmente congiunzioni (o disgiunzioni).

Sincronizzazione: Una label di tipo Sincronizzazione può contenere espressioni del tipo *Expression!* oppure *Expression?*. Le espressioni devono essere side-effect free, valutate sul canale, e referenziare dati di tipo intero, clock o canali.

Update: La label Update contiene una lista di espressioni di tipo side-effect. Le espressioni sono applicate ad interi, clock e costanti, ed è possibile assegnare ad un clock solo valori interi. Inoltre viene fornita la possibilità di richiamare le funzioni.

Invariante: Un invariante è un espressione di tipo side-effect free, valutata generalmente su un clock x nella forma $x < e$ o $x \leq e$, che può contenere dati di tipo intero, costanti o clock. La condizione deve essere sempre vera affinché la locazione sia raggiungibile dall'automa, sia in entrata che in uscita [10] [6] [8] [11].

1.1.3 Il Model Checking Statistico

Il Model Checking Statistico è una tecnica di validazione introdotta recentemente, che ha l'obiettivo di consentire la verifica delle proprietà in logica temporale di un modello non deterministico, per un sistema complesso.

L'idea alla base di SMC è quella di effettuare un determinato numero di simulazioni del sistema per poter effettuare una stima statistica della probabilità con cui si verifica una certa proprietà, con un elevato grado di confidenza. I risultati ottenuti vengono in seguito utilizzati per poter progettare ed implementare correttamente il sistema sul quale si sta lavorando. I vantaggi di progettare un sistema utilizzando SMC, risiedono nella facilità d'uso, implementazione e comprensione (specie in ambito industriale, ingegneria del software, ecc.). Inoltre il suo formalismo è risultato essere molto efficiente.

Sia M un automa non deterministico, e sia φ una proprietà da verificare. Definiamo con P_M la probabilità con la quale viene soddisfatta la proprietà per l'automa M.

1. **Probability Estimation**: Qual'è la probabilità P_M che si verifichi una proprietà φ per l'automa M ?

2. **Hypothesis Testing**: La probabilità P_M per un automa M è maggiore o uguale ad un valore di soglia $p \in [0, 1]$?

3. **Probability Comparison**: La probabilità $P_M(\varphi_1)$ è maggiore della probabilità $P_M(\varphi_2)$?

Da un punto di vista concettuale, rispondere a questi quesiti utilizzando SMC è semplice. Ogni simulazione del sistema è codificata come una variabile casuale di Bernoulli, la quale risulta vera se la simulazione soddisfa la proprietà, in caso contrario falsa.

Probability Estimation: L'algoritmo per la stima della probabilità, calcola il numero dei *runs* necessari, al fine di ottenere un intervallo di approssimazione $[p-\varepsilon, p+\varepsilon]$ per $p = \mathbb{P}(\psi)$ con una confidenza pari a $1-\alpha$.

Hypothesis Testing: Sia $p = P(\varphi)$, per determinare se $p \geq \theta$, possiamo effettuare una verifica $H : p = P_M(\varphi) \geq \theta$ contro le ipotesi alternative $K : p = P_M(\varphi) < \theta$. Per limitare la probabilità di fare errori, si possono scegliere due parametri α e β e verificare le ipotesi $H_0 : p \geq p_0$ e $H_1 : p \leq p_1$ con $p_0 = \theta + \delta_0, p_1 = \theta - \delta_1$. L'intervallo $p_0 - p_1$ denota la regione di indifferenza con p_0, p_1 usati come valori di soglia nell'algoritmo. Il parametro α è la probabilità di accettazione di H_0 quando viene soddisfatta H_1 (falsi positivi) e β è la probabilità di accettazione di H_1 quando viene soddisfatta H_0 (falsi negativi).

Probability Comparison: L'algoritmo è un'estensione di quello che viene utilizzato in *Hypothesis Testing* (Test di Wald) [9] [12] [1].

1.1.4 Introduzione alle VANET

In letteratura, una **VANET** è una rete costituita da veicoli abilitanti a comunicare tra loro attraverso lo scambio di informazioni rilevate dall'*On Board Unit* (**OBU**) installata a bordo del veicolo medesimo. I dati vengono successivamente trasmessi (o ricevuti), utilizzando protocolli di comunicazione wireless a corto raggio. Uno dei protocolli maggiormente utilizzati che consente di creare reti mobili è l'*IEEE 802.11p*.

I nodi (chiamati *Agenti*) sono rappresentati dai veicoli. Nelle VANET ci sono principalmente due tipologie di comunicazioni: *Vehicle-to-Vehicle* (**V2V**) e *Vehicle-to-Infrastucture* (**V2R**). Nella comunicazione V2V, i veicoli trasmettono messaggi tra loro. Nella comunicazione V2R invece, vengono utilizzate le unità a bordo strada, le cosiddette *Road Side Unit* (**RSU**), al fine di trasmettere informazioni rilevanti, ai veicoli.

L'implementazione di una VANET porta numerosi benefici, tra cui la segnalazione di emergenze, anomalie, condizioni sul traffico e parametri ambientali.

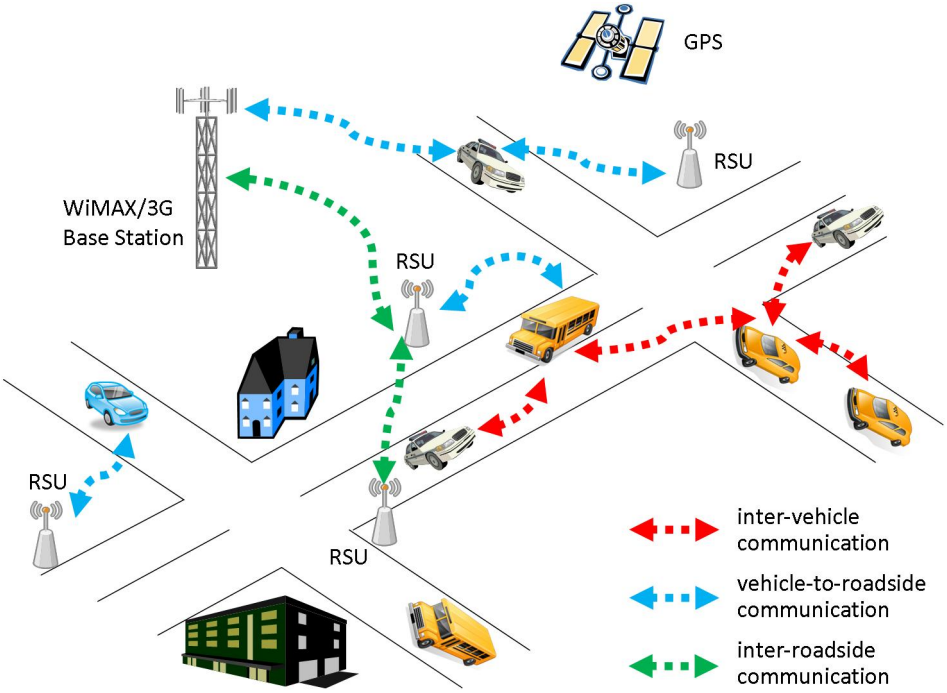


Figura 1: Vehicle-to-Vehicle e Vehicle-to-Infrastucture

Alcune proprietà rilevanti delle reti veicolari sono: *scalabilità efficiente, risorse limitate dei dispositivi, rapidità di trasmissione in broadcast, monitoraggio del traffico, mobilità, robustezza* e ***disseminazione***.

La disseminazione consente ad un veicolo in procinto di percorrere un determinato tratto stradale, di ricevere con largo anticipo informazioni su quest'ultimo, consentendo quindi una prevenzione di situazioni anomale. In una comunicazione V2V nella maggior parte delle volte si utilizza un modello di comunicazione per l'invio delle informazioni denominato *Push-Model*. Questo modello prevede che l'invio delle informazioni venga effettuato in broadcast, così da evitare l'utilizzo di algoritmi di routing.

L'obiettivo consiste nel raccogliere le informazioni dal veicolo sender e di inviarle a tutti quei nodi che si trovano nel raggio di trasmissione del veicolo mittente. Il meccanismo di disseminazione permette di inviare contemporaneamente i dati generati del veicolo stesso (***Generated Data***) e i dati ricevuti dai veicoli vicini (***Relayed Data***) con una tempistica periodica dettata dall'uso del *Broadcast Period* (BP). Ciò significa che alla ricezione di un pacchetto di informazioni, il suo inoltro viene posticipato al successivo periodo di broadcast. Considerando un veicolo V, è possibile analizzare quali sono i modelli di disseminazione:

1. **Same Direction (Same-dir)**: la disseminazione dei dati avviene lungo la direzione di marcia del veicolo V.

2. **Opposite Direction (Opposite-dir)**: la disseminazione dei dati avviene utilizzando i veicoli che viaggiano in direzione opposta rispetto a V.

3. **Bi-Direction (Bi-dir)**: la disseminazione dei dati avviene utilizzando i veicoli in entrambe le direzioni.

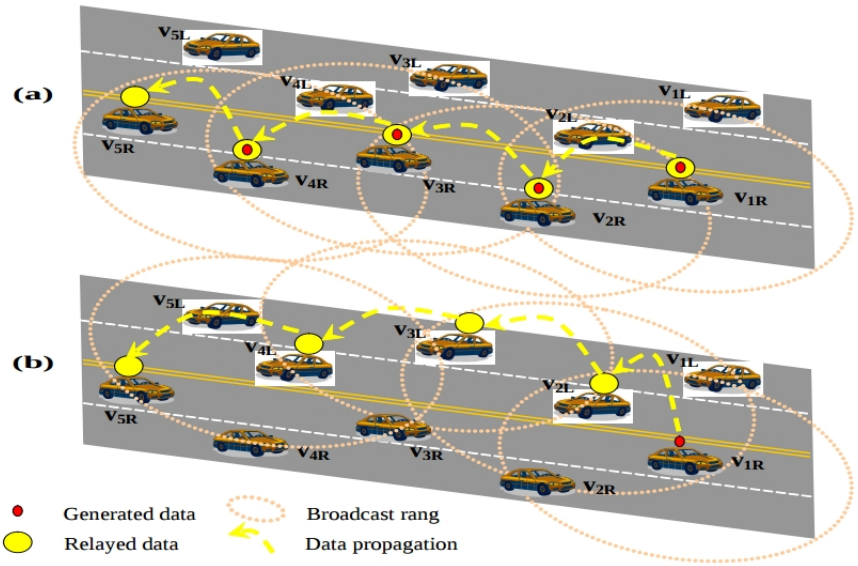


Figura 2: Modelli di disseminazione: a) Same-dir, b) Opposite-dir

Nel modello Opposite Direction, i Generated Data del veicolo V vengono propagati indietro da quei veicoli che viaggiano in direzione opposta, i quali a loro volta li replicheranno come Realyed Data. Assumiamo che un veicolo A invii un pacchetto, ed un veicolo B sia nel suo raggio di trasmissione:

1. Se A e B si muovono da sinistra verso destra, B accetterà il pacchetto se A si trova a destra rispetto a B. Il veicolo A invierà in broadcast i suoi Generated Data.

2. Se A e B si muovono da destra verso sinistra, B accetterà il pacchetto se B si trova a sinistra rispetto a A. Il veicolo A invierà i Relayed Data.

3. Se A e B si muovono in direzioni opposte, B accetterà il pacchetto indipendentemente dalle posizioni dei veicoli.

La disseminazione inoltre è determinata dal veicolo *receiver*, cioè sono i veicoli stessi a stabilire se un pacchetto ricevuto è accettabile o meno.

2 Realizzazione del modello

Il modello di disseminazione Opposite direction è stato realizzato principalmente mediante la definizione di due automi:

1. **Car**: l'automa descrive il modello di comunicazione ed il modello di movimento dei veicoli;

2. **RSU**: l'automa si occupa di gestire il modello di comunicazione delle unità a bordo strada.

Nelle dichiarazioni globali, sono state definite le specifiche per la costruzione della mappa, i dettagli realizzativi per il modello di trasmissione, le impostazioni iniziali dei veicoli, e la dichiarazione di due funzioni (una per il calcolo del valore assoluto, ed una per valutare la distanza tra due punti sulla mappa mediante la distanza di Manhattan). Una particolarità dell'automa RSU risiede nel fatto che le istanze del template vengono generate dinamicamente, e pertanto il template stesso deve essere dichiarato come *dinamico* (in Uppaal si utilizza la parola chiave

dynamic). Il funzionamento all'interno del sistema delle istanze dinamiche è identico a quello delle statiche, con l'eccezione che esse possono essere rimosse dal sistema, in un qualsiasi istante di tempo **t**, terminando di conseguenza la loro esecuzione.

```
/**INIZIO Funzione ABS*/
int abs(int value)
{
if (value<0) value=-value; return value;
}
/**FINE Funzione ABS*/

/**INIZIO Distanza di Manhattan*/
int distance(int x1,int y1, int x2, int y2)
{
return abs(x1-x2) + abs(y1-y2);
}
/**FINE Distanza di Manhattan*/

dynamic RSU(const id_i idR, int idc);
```

Listato 1: Funzione ABS - Distanza di Manhattan - Template RSU dinamico

Il vantaggio in questo caso consiste nel fatto che gli automi per gestire le RSU vengono istanziati laddove è necessario, cioè lí dove si verifica l'anomalia di un determinato veicolo. Di conseguenza dal punto di vista prestazionale si ha un notevole risparmio di memoria ed un uso efficiente delle risorse del sistema consentendo quindi una praticabilità proficua del model checking, relativa alla gestione dell'**esplosione combinatoria degli stati** [10] [3] [5] [7] [4] [2].

2.1 Analisi e descrizione generale del modello

Il modello descrive uno scenario con raccordi tra tratti a rapido/lento scorrimento, ed il centro città. Inoltre sono state definite strade a 2 e a 4 corsie con i rispettivi incroci. I veicoli inseriti nelle simulazioni hanno capacità prestazionali definite in fede ai limiti di velocità sui tratti stradali impostati nel modello.

```
const int Nc=15;
const int Ni=10;
const int Ns=13;

clock time;

typedef int[0,Nc-1] id_c;
typedef int[0,Ni-1] id_i;
typedef int[0,Ns-1] id_s;

int [0,Nc] count=0;
int settings;

const int speedmax_way=40;
const int slowmax_way=15;
```

```
16 const int dist=100;
```

Listato 2: Proprietà globali del sistema

Le variabili *Nc*, *Ni* ed *Ns* (dichiarate come costanti numeriche), rappresentano rispettivamente il numero dei **veicoli**, **intersezioni** e **segmenti**. Successivamente mediante *typedef* sono stati definiti i nomi alternativi alle tipologie di dato dichiarate, e quindi un identificativo per ogni veicolo, intersezione e segmento. Inoltre è stato dichiarato un parametro **time** di tipo *clock* per tenere in considerazione il tempo trascorso durante l'intera simulazione.

La variabile **settings** è stata dichiarata per controllare se la totalità dei veicoli è stata configurata prima che vengano eseguite le verifiche sulla possibile ricezione di un messaggio di anomalia. La variabile **count** è una *progress measure*, cioè un'espressione che consente di misurare il progresso del sistema. Nel nostro caso, è stata usata per ricavare quanti sono i veicoli che durante una simulazione riescono a ricevere il messaggio di anomalia.

Questa è un'ottimizzazione rilevante in termini di costi computazionali, poichè dato che per ogni stato del sistema si deve valutare e calcolare questo parametro, ciò può risultare altamente dispendioso. Successivamente sono state definite le velocità per i segmenti a *rapido/lento* scorrimento espresse in m/s e la distanza necessaria a verificare la probabilità che una percentuale di veicoli riceva la segnalazione di una anomalia.

2.1.1 Rappresentazione dello scenario

La figura sottostante illustra la mappa nella quale avviene la simulazione dell'intero sistema.

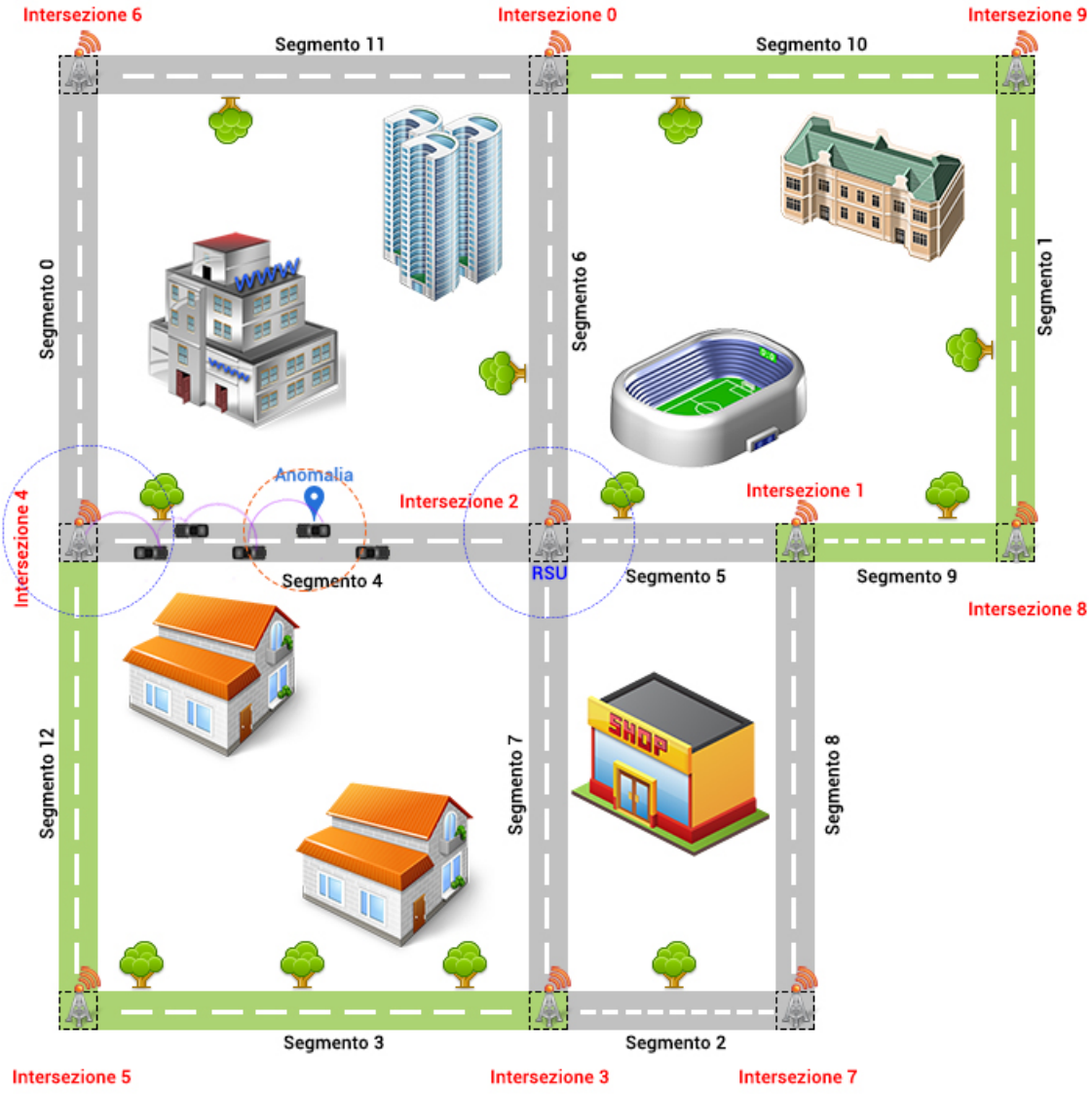


Figura 3: Mappa dello scenario

La mappa è stata costruita pensando ad essa come l'insieme di 10 (Ni) intersezioni in cui convergono i 13 (Ns) segmenti stradali. L'intero scenario misura 2 km^2 ed è stato definito mediante strutture matriciali.

```
1 const int Map[id_i][4]= {
2 {6,10,11,-1},
3 ...
4 };
```

Listato 3: Matrice Intersezioni-Segmenti

Nella struttura matriciale **Intersezioni-Segmenti**, il parametro *id_i* rappresenta l'id dell'intersezione, mentre il parametro 4, il massimo numero dei possibili segmenti che può convergere nell'intersezione stessa. Ad esempio Map[0][4]={6,10,11,-1} definisce l'intersezione 0, dalla quale si può accedere ad uno dei seguenti segmenti: 6,10,11. Il valore -1 invece, distingue l'assenza di un segmento.

```
const int nInters[id_s][2]={
{6,4},{9,8},
...
};
```

Listato 4: Matrice Segmento-Intersezioni

La matrice **Segmento-Intersezioni** è stata costruita tenendo conto della direzione di percorrenza di un veicolo. Nello specifico, *id_s* rappresenta l'identificativo del segmento, mentre il parametro 2 serve a definire il numero dei segmenti, in particolare a identificare quale sarà la prossima intersezione che si incontrerà se si sta percorrendo il tratto stradale verso destra o verso sinistra. Ad esempio nInters[0][2]={6,4} è il segmento 0; se quest'ultimo viene percorso dal veicolo verso destra, la prossima intersezione sarà la 6, viceversa la 4.

```
typedef struct
{
    int Px;
    int Py;
} Int;

Int I[Ni]={
{1000,1000},
...
};
```

Listato 5: Coordinate Intersezioni

Per quanto concerne le **Coordinate delle Intersezioni**, vengono istanziate *Ni* strutture di tipo **Int**, poichè *Ni* sono le intersezioni alle quali dobbiamo assegnare le rispettive coordinate:

- Px: coordinata x dell'intersezione
- Py: coordinata y dell'intersezione

Ad esempio, l'intersezione I[0] avrà coordinate (1000,1000).

```
typedef struct
{
int length;
int nc_l;
int nc_r;
bool expressway;
} Str;

Str S[Ns]= {
```

```
{600,0,0,0 },
...
};
```

Listato 6: Proprietà del segmento

Per poter definire le **Proprietà di un Segmento** è stata definita una struttura **Str** che determina la lunghezza del segmento *length*, il numero dei veicoli nella corsia sinistra (*nc_l*) e destra (*nc_r*). La tipologia del tratto viene specificata mediante una variabile booleana *expressway*, la quale vale 1 se quest'ultimo è a scorrimento veloce, 0 viceversa.

2.1.2 Parametri del modello di trasmissione

Il modello di trasmissione dei veicoli prevede che siano dichiarati *Nc* canali di broadcast, tanti quanti sono i veicoli (ogni veicolo ha un canale di trasmissione) ed *Ni* canali per le RSU. Inoltre sono state definite le due tipologie di messaggio (Generated o Relayed).

```
broadcast chan msg[Nc]; broadcast chan alert[Ni];

const int GEN=2; const int REL=3;

/*Contenuto del messaggio trasmesso*/
typedef struct
{
int type;
...
int VxA; int VyA;
} Msg;

Msg Mgen[Nc]; Msg Mrec[Nc]; Msg RSUAnomaly[Ni];
int RR=400;
```

Listato 7: Parametri del modello di trasmissione

Nella struttura del messaggio, è stata specificata la tipologia del messaggio trasmesso, le coordinate, il segmento corrente e la direzione di marcia del veicolo, l'eventuale segnalazione di anomalia (0 se negativo, 1 se positivo) e infine le coordinate dell'anomalia.
Inoltre sono stati definiti i buffer per gestire i messaggi da trasmettere e quelli ricevuti, sia per i veicoli, che per le unità a bordo strada, ed il raggio trasmissione dell'RSU.

2.1.3 Parametri del veicolo

Il veicolo invece possiede le proprietà relative al raggio di trasmissione, *BP* e un set di velocità: minima, media e massima.

```
const int r=250;
const int slow_speed=6;
const int mean_speed=20;
const int high_speed=50;
const int BP=7;
```

Listato 8: Parametri del veicolo

2.2 Modello del veicolo

In questa sezione verrà descritto il modello dell'automa relativo ai nodi mobili presenti nella VANET, in base alle ipotesi teoriche precedentemente illustrate.

2.2.1 Descrizione dell'automa

Per realizzare l'automa relativo al veicolo, sono state innanzitutto individuate le azioni principali, o fasi, che il veicolo svolgerà all'interno della nostra rete. Queste fasi consistono in:

1. Inizializzazione e settaggio dei parametri
2. Ricezione dei messaggi
3. Invio in broadcast dei messaggi
4. Cambio del segmento e aggiornamento della posizione

Analizziamo dunque, nel dettaglio, le varie fasi.

Inizializzazione e settaggio dei parametri: Questa fase si compone di 3 locazioni di tipo *Committed* (compresa la locazione iniziale), utili per modellare le operazioni atomiche. Nella prima transizione ciascun veicolo sceglie, attraverso 2 Select, l'**ID** del **segmento** ed un valore intero tra [0,1], che rappresenterà la direzione di marcia (destra/verso l'alto oppure sinistra/verso il basso). Tali parametri saranno passati alla funzione *setParameters()*, che descriveremo nel dettaglio nella prossima sezione.

Nella seconda transizione vengono inizializzate le velocità di partenza dei veicoli, in base al segmento scelto, al numero di veicoli presenti nel segmento stesso e al tipo di segmento (a scorrimento veloce o lento). La guardia *settings=NC* assicura che tale transizione sia effettuata, da tutti veicoli, dopo che essi abbiano effettuato regolarmente la scelta del segmento. In questo modo si assicura una corretta scelta delle velocità, poichè risulta essenziale conoscere il numero dei veicoli presenti nella corsia. In questa fase inoltre, viene scelta anche la posizione del veicolo all'interno del segmento, secondo la relazione $posIn = segmentLength/(idCar + 1)$, tale da assicurare una posizione di partenza diversa per ogni veicolo, anche nel caso in cui si trovino nello stesso segmento.

Infine nell'ultima transizione vengono effettuati i controlli relativi al veicolo che ha generato l'anomalia. Le guardie sulla variabile *isAnomaly*, settata nella **Fase 1** (nel nostro caso si è scelto di assegnarla al veicolo con $ID = 9$), assicurano che soltanto il veicolo che ha generato l'anomalia, istanzi a run time gli automi di tipo RSU, relativi alle intersezioni del segmento corrente, tramite la funzione *spawn()*.

Successivamente l'automa raggiunge la locazione **Car**, dalla quale potrà eseguire le prossime azioni del sistema. In questa locazione è presente l'invariante $x <= tCross\ and\ x <= BP$, dove **tCross** è il tempo necessario per attraversare il segmento, calcolato sia inizialmente che in fase di aggiornamento delle coordinate, e **BP** è il *Broadcast Period*. Le condizioni, affinchè l'automa possa restare nella locazione, devo essere verificate contemporaneamente: la condizione sul BP assicura che ogni BP secondi venga attraversato l'edge relativo all'invio del messaggio, mentre la condizione su tCross permette di effettuare il cambio di segmento una volta raggiunta la fine del segmento corrente. Sostanzialmente, poichè si è effettuato il ricalcolo del tCross ad ogni BP, la condizione può essere intrerpreta come $t = min(tCross, BP)$: ad un certo istante di tempo,

in prossimità dell'incrocio, il tempo di attraversamento sarà inferiore rispetto al BP, garantendo una corretta esecuzione temporale delle fasi dell'automa.

Ricezione dei messaggi: In questa fase abbiamo 2 transizioni, le quali terminano sulla locazione Car, relative alla ricezione dei messaggi da parte di altri **nodi mobili** o di una **RSU**. La guardia $x < BP$, presente in entrambi i casi, limita la possibilità di attraversare gli edge (e quindi di ricevere un messaggio) in istanti di tempo compresi tra $[0, BP[$. Attraverso le Select viene selezionato l'ID del nodo mobile o della RSU che sta trasmettendo un messaggio, e tramite le funzioni *inRange()* e *inRangeRSU()* si verifica, rispettivamente, che il ricevente si trovi nel raggio di trasmissione del sender e rispetti la posizione reciproca tra i veicoli, secondo il modello di disseminazione. La sincronizzazione avviene tramite i canali *alert* e *msg*. Tramite le funzioni *receive()* e *receiveRSU()* viene infine effettuato l'aggiornamento del messaggio e la registrazione dell'anomalia ricevuta. E' utile precisare che, in entrambi i casi, la ricezione del messaggio avviene soltanto se non sia stata precedentemene ricevuta, al fine di evitare un overhead di rete e messaggi ridondanti.

Invio in broadcast dei messaggi: Per questa azione è stata prevista una locazione **Send**, di tipo *Committed*, utile per distinguere l'invio di Generated o Relayed Data. In maniera speculare alla ricezione, l'invio dei messaggi avviene, combinando l'invariante di Car con la guardia $x == BP$, esattamente ogni BP secondi. La funzione *send()* setta un Generated Data con le informazioni attuali del veicoli, ed eventualmente include nel messaggio le coordinate del punto di anomalia ricevute negli istanti precedenti. Inoltre, prima di effettuare questa operazione, aggiorna le coordinate del veicolo, in base al senso di marcia e del segmento, considerando che in un BP il veicolo percorre esattamente $BP * v_c$ metri, dove v_c è la velocità corrente. La sincronizzazione avviene sempre sul canale *msg!*. Nella stessa funzione avviene anche il ricalcolo di **tCross** e l'azzeramento del clock x.

Successivamente, nello stato Send, avviene la verifica di un eventuale messaggio di relay da inoltrare, mediante la funzione *relayReady()*, inserita come guardia. Nel caso positivo avviene un ulteriore invio immediato di un pacchetto di tipo Relay, in caso negativo non viene effettuata nessuna azione, e l'automa ritorna nello stato Car.

Cambio del segmento e aggiornamento della posizione: L'azione, relativa al cambio del segmento, avviene contestualmente alla violazione dell'invariante dello stato Car, condizione che ha come significato il raggiungimento di un incrocio. Viene selezionato casualmente l'ID del prossimo segmento nel quale immettersi, dalla lista dei segmenti connessi all'intersezione, controllando, tramite la funzione *segEnabled()* presente nella guardia, che esso non corrisponda al segmento corrente, o che abbia valore -1, cioè non esiste un segmento connesso. In base alla scelta effettuata, si effettua l'aggiornamento della posizione, della direzione di marcia, l'azzeramento del clock e il ricalcolo del tempo di attraversamento per il nuovo segmento. Viene inoltre aggiornato il valore del numero dei veicoli per i segmenti in uscita ed in ingresso, utile per il ricalcolo delle velocità.

La guardia ***!isAnomaly***, presente nelle transizioni relative all'invio, alla ricezione, ed al cambio segmento, è stata inserita per consentire di effettuare tali operazioni solo ai veicoli che non hanno generato l'anomalia. Viceversa, il veicolo che ha generato l'anomalia effettuerà come operazione il broadcast costante del messaggio di allerta, grazie al settaggio di tCross pari a BP per tutti gli istanti di tempo.

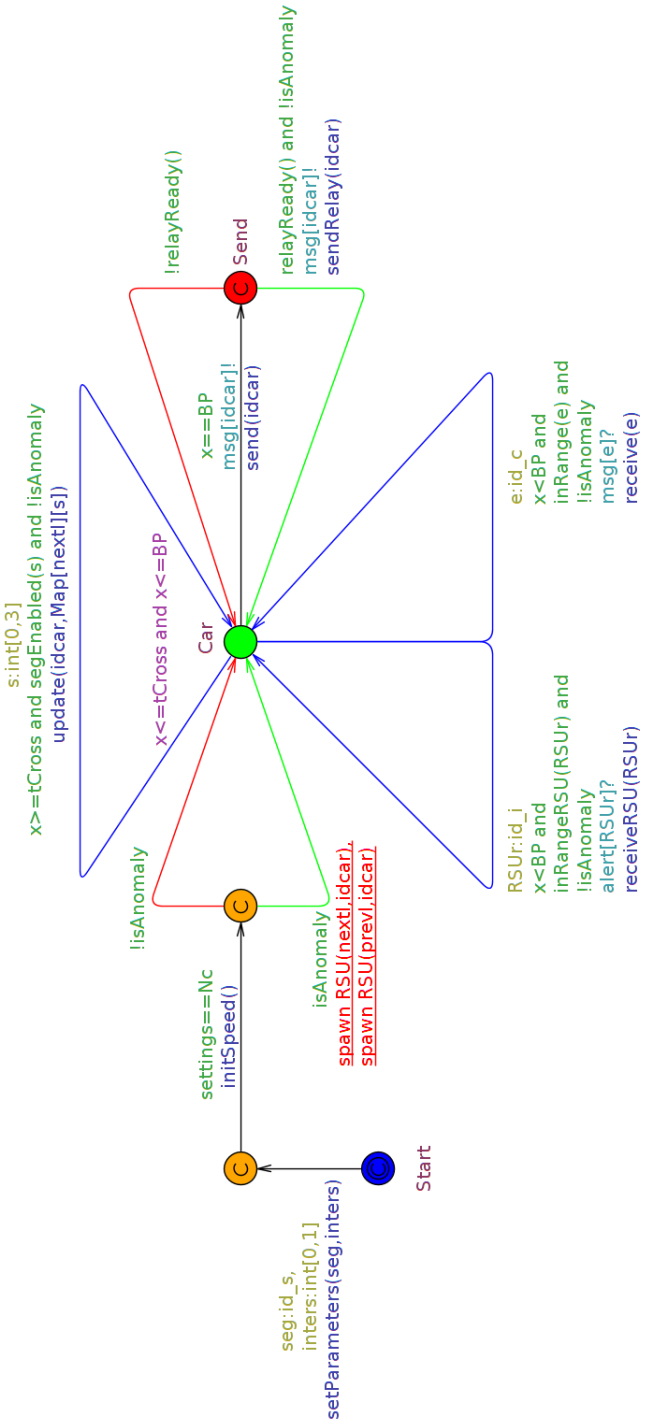


Figura 4: Automa Car

2.2.2 Descrizione del codice

In questa sezione saranno descritti i metodi utilizzati per la classe Car, esplicitando i loro prototipi ed i parametri.

```
1 clock x;
2
3 int cur_speed;
4
```

```
5  bool dir;
6  int cur_seg;
7  int Vx,Vy;
8  int VxA=-1;
9  int VyA=-1;
10 int tCross;
11 int crossed;
12 int posIn;
13 int nextI;
14 int prevI;
15
16
17 bool isAnomaly;
18 bool anomaly_rcv=false;
```

Listato 9: Variabili dell'automa Car

Il template **Car** viene parametrizzato mediante il parametro *idcar* (identificativo locale del veicolo) di tipo *id_c* (tipo identificativo che rappresenta i veicoli in generale). Per ogni veicolo vengono definite le seguenti proprietà:

- Un Clock per poter gestire le tempistiche del veicolo
- Velocità corrente, Direzione, Segmento corrente, Coordinate (posizione e anomalia) del veicolo
- Tempo per percorrere un segmento, metri percorsi, posizione iniziale del veicolo, prossima e precedente intersezione del veicolo
- Variabile *isAnomaly* per verificare quando il veicolo si trova nello stato di anomalia, e *anomaly_rcv* inizialmente impostata a **false**, al fine di verificare l'avvenuta ricezione di un'anomalia.

Le coordinate dell'anomalia assumeranno valore -1 fino a quando il veicolo non entrerà nello stato di **Anomalia**.

```
1 void setParameters(int seg, int d)
```

Listato 10: Metodo setParameters

Parametri:

- **seg** è l'identificativo del segmento selezionato
- **d** è la direzione di marcia selezionata

Il metodo *setParameters()* viene utilizzato per settare i parametri dei veicoli nella fase iniziale. In particolare, dal parametro **d** vengono settate **prevI** e **nextI** (intersezione precedente e successiva rispettivamente), selezionandone il valore dalla matrice **nInters**. Le coordinate (Vx, Vy) in questa fase sono poste uguali al punto relativo all'intersezione precedente. In questa fase si determina il veicolo responsabile dell'anomalia: in particolare, il veicolo con **ID=9** imposterà a **true** il valore di **isAnomaly**. Infine, in base della direzione viene incrementato il numero dei veicoli nella corsia del segmento corrispondente.

```
1 void setSpeed()
```

Listato 11: Metodo setSpeed

La funzione *setSpeed()*, consente di regolare la velocità del singolo veicolo tenendo in considerazione il limite massimo di velocità raggiungibile sul tratto stradale, il numero dei veicoli presenti sulla carreggiata e la velocità massima raggiungibile dal veicolo. Ci limitiamo a descrivere i controlli effettuati sulla velocità del veicolo relativi carreggiata destra, poiché questi risultano essere identici a quelli effettuati sulla carreggiata sinistra.

Se il veicolo sta procedendo verso destra ed il numero dei veicoli sulla carreggiata è diverso da zero, la velocità corrente sarà settata come il rapporto pari alla massima velocità raggiungibile del veicolo ed il numero dei veicoli presenti. Nello specifico si va ad analizzare se il segmento stradale è a rapido o lento scorrimento e se la velocità del veicolo è superiore o inferiore al limite specificato.

In entrambi i casi, si andrà a definire la velocità del veicolo mediante il rapporto della velocità massima (minima) raggiungibile sul tratto stradale, ed il numero dei veicoli presenti su di esso. Infine se il veicolo possiede una velocità corrente inferiore alla velocità minima che questo dovrebbe avere, la velocità corrente sarà settata pari a quest'ultima precedentemente definita.

```
1 void initSpeed()
```

Listato 12: Metodo initSpeed

Il metodo *initSpeed()* imposta la velocità iniziale, il tempo di attraversamento del segmento iniziale, ed il messaggio generato per ogni veicolo. In primo luogo viene richiamata la funzione **setSpeed()**, la quale determina la velocità in base al numero di veicoli presenti nella corsia.

Successivamente viene determinato un offset (**posIn**) della posizione per ogni veicolo, all'interno del segmento. Esso risulta essere univoco, poiché viene calcolato utilizzando l'ID del veicolo. Le coordinate iniziali vengono aggiornate sulla base di questo offset, considerando sia la direzione di marcia che lo spostamento sull'asse x o y: quest'ultimo in particolare è determinato confrontando le coordinate di **prevI** e **nextI**, considerando che se lo spostamento avviene sull'asse x, la coordinata x per le intersezioni è uguale, e viceversa.

Il tempo di Cross, in questa fase, viene impostato considerando l'offset assegnato, e dunque i metri idealmente già percorsi dal veicolo. Infine, viene inizializzato un messaggio generato con le informazioni relative alle coordinate, al segmento ed alla direzione. Il veicolo che ha generato l'anomalia, inoltre, setta nel messaggio generato le informazioni relative al punto di anomalia (corrispondente alla posizione iniziale).

```
1 void update(int id,int ns)
```

Listato 13: Metodo Update

La funzione *Update()*, è stata definita per consentire l'aggiornamento del prossimo segmento in cui il veicolo dovrà immettersi.

Parametri:

- **id** è il l'identificativo del veicolo
- **ns** è il segmento in cui è situato il veicolo

In primis, si va a memorizzare il segmento corrente in una variabile temporanea di tipo **meta**. Il tipo **meta** viene usato per effettuare lo swapping di due variabili intere; esse vengono memorizzate nel vettore degli stati, ma semanticamente non vengono considerate come parte dello stato (ciò si traduce in un risparmio in termini di memoria per il sistema).

Successivamente andiamo a decrementare dal numero dei veicoli presenti sulla carreggiata, il veicolo che sta uscendo del segmento corrente, per poi in seguito inserirlo nel prossimo segmento sul quale si immetterà. Verifichiamo se il primo elemento di **nInters** coincide con l'intersezione precedente del veicolo. Se questo è verificato vuol dire che il veicolo sta procedendo verso sinistra, altrimenti verso destra (questa deduzione deriva per costruzione dalla struttura **Matrice Segmento-Intersezioni**). In base a ciò, si imposta la nuova direzione e la nuova intersezione del veicolo, con il conseguente aggiornamento del numero dei veicoli attualmente presenti sul nuovo segmento.

Infine vengono assegnate le nuove coordinate al veicolo, pari a quelle della nuova intersezione; in seguito viene azzerato il clock, dato che il veicolo è entrato nel nuovo segmento, ed successivamente si procede al ricalcolo del tempo di percorrenza del tratto, mediante il rapporto tra la lunghezza del segmento e la velocità corrente del mezzo.

```
1 void send(int id)
```

Listato 14: Metodo Send

Parametri:

- **id** è il l'identificativo del veicolo

Il metodo *send()*, richiamato ad ogni BP, si occupa di aggiornare le informazioni del veicolo riguardo posizione, tempo di cross e velocità, sia all'interno delle variabili locali che nel messaggio generato. Tramite la variabile booleana **anomaly_rcv**, si verifica se nel BP precedente sia stata ricevuta una segnalazione di un'anomalia, ed in caso affermativo si popola il messaggio generato da inviare con le informazioni relative all'anomalia stessa. L'arrivo del veicolo ad un incrocio comporta necessariamente una decelerazione da parte dello stesso: per questo motivo, se il veicolo sta percorrendo gli ultimi 100 metri del tratto di strada, imposta come velocità corrente un valore pari a **slow_speed**, tale da simulare l'attraversamento dello stesso.

```
1 void sendRelay(int id)
```

Listato 15: Metodo sendRelay

Setta il messaggio di tipo Relayed.

```
1 bool inRange(int id_tx)
```

Listato 16: Metodo inRange

Parametri:

- **id_tx** è l'identificativo del veicolo sender

Il metodo *inRange()*, inserito come guardia, si occupa di abilitare o meno l'edge relativo alla transizione che porta alla ricezione di un messaggio, secondo il modello di disseminazione Opposite Direction. Il primo controllo effettuato prevede la verifica delle seguenti condizioni:

1. Sender e Receiver devono trovarsi nello stesso segmento

2. Il messaggio generato deve contenere la segnalazione di un'anomalia

3. Il veicolo non ha ancora ricevuto una segnalazione di anomalia

Una volta verificate le condizioni preliminari, si analizzano tutti i casi possibili a seconda delle posizioni reciproche tra i veicoli comunicanti, includendo in ogni caso la condizione sul raggio di trasmissione.

Valutando il segno della differenza $d = V - Mgen[id\_tx].Vx$ (o su y, $.Vy$), è possibile valutare se il veicolo sender precede il receiver nella stessa corsia. Nel primo caso si trasmette un Generated Data ($d < 0$), viceversa nel caso di Relayed Data, nella stessa corsia ($d > 0$). Se uno di questi casi è verificato la funzione restituisce **true** ed abilita l'edge per la ricezione.

```
1 void receive(int id_tx)
```

Listato 17: Metodo receive

Parametri:

- **id_tx** è l'identificativo del veicolo sender

Mediante questo metodo, vengono effettuate le operazioni di aggiornamento sulla base dell'anomalia ricevuta. Inoltre, se la distanza tra la posizione corrente del veicolo ed il punto di anomalia è maggiore o uguale al valore **dist**, viene incrementato il valore di **count**. Nel caso di ricezione di un Generated Data da un veicolo nella corsia opposta, oppure di un Relayed Data nella stessa corsia, viene settato un messaggio con le informazioni dell'anomalia, da inoltrare come Relayed al prossimo BP.

```
1 bool inRangeRSU(int id_r)
```

Listato 18: Metodo inRangeRSU

Parametri:

- **id_r** è l'identificativo della RSU

La funzione *inRangeRSU()*, viene abilitata quando un veicolo riceve un messaggio di anomalia da un'unità a bordo strada. I controlli effettuati, in questo caso, richiedono che il veicolo non abbia precedentemente ricevuto nessuna segnalazione, e che si trovi nel raggio di trasmissione della RSU. Tale distanza viene calcolata utilizzando la funzione **distance** (distanza di Manhattan).

```
1 bool relayReady() {return Mrec[idcar].VxA == -1 and Mrec[idcar].
    VyA == -1 ? false:true;}
```

Listato 19: Metodo relayReady

Controlla se è presente nel buffer un Relayed Data da inoltrare.

```
1 bool segEnabled(int id_seg) { return Map[nextI][id_seg]!=-1 and
    Map[nextI][id_seg]!=cur_seg ? true:false;}
```

Listato 20: Metodo segEnabled

Parametri:

- **id_seg** è l'identificativo del prossimo segmento

La funzione *segEnabled()*, utilizzata come guardia, ha lo scopo di verificare se l'ID del prossimo segmento, selezionato in maniera casuale, è valido. Ciò avviene controllando sia se l'elemento scelto appartiene alla lista dei segmenti connessi all'intersezione successiva ($! = -1$), sia che non corrisponda al segmento attuale (il segmento attuale è sicuramente connesso alla prossima intersezione, quindi va scartato). In caso di esito positivo restituisce **true**, abilitando di conseguenza l'egde.

2.3 Modello dell'RSU

In questa sezione sarà descritto, in dettaglio, il modello dell'automa relativo alle unità a bordo strada (RSU), dal punto di vista strutturale e del codice implementato.

2.3.1 Descrizione dell'automa

La struttura realizzata per il modello delle RSU è formata dai seguenti elementi:

- Una variabile di tipo clock **y**
- Locazione **Idle**: E' la locazione di partenza, e prevede, nel caso di attivazione dell'automa tramite la funzione *spawn()*, un'uscita immediata per settare i parametri della RSU.
- Locazione **Anomaly**: Raggiunta tale locazione, l'unità a bordo strada inoltra il messaggio di anomalia ricevuto, sincronizzandosi come sender sul canale **alert!** ogni BP secondi.

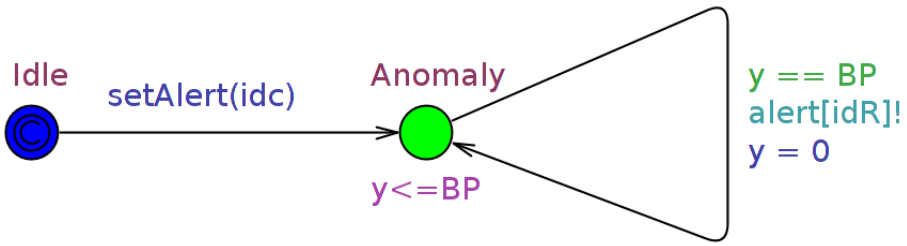


Figura 5: Automa RSU

2.3.2 Descrizione del codice

Nella parte relativa al codice, oltre al clock descritto in precedenza, è stato dichiarato il metodo *setAlert()*, il cui codice sorgente è mostrato di seguito.

```
1 void setAlert(int idc)
2 {
3     RSUAnomaly[idR].Vx = I[idR].Px;
4     RSUAnomaly[idR].Vy = I[idR].Py;
5     RSUAnomaly[idR].VxA = Mgen[idc].VxA;
6     RSUAnomaly[idR].VyA = Mgen[idc].VyA;
7 }
```

Listato 21: Metodo setAlert

Il parametro **idc** rappresenta l'ID del veicolo che ha generato l'anomalia, dal quale l'RSU estrapolerà i dati necessari alla sua inizializzazione. In particolare, attraverso tale ID esso accederà al messaggio generato dal veicolo, memorizzando le coordinate VxA e VyA all'interno della struct **RSUAnomaly[idR]**.

3 Conclusioni e Risultati

Nella fase finale del progetto, sono state verificate le proprietà relative al modello realizzato, con l'aiuto del tool SMC di Uppaal . La verifica sulla percentuale dei veicoli è stata effettuata monitorando la *Progress Measure* **count**, suddividendo le query in set, in base alla distanza richiesta (100,300,500,700,1000) espressa in metri. Per ogni set di valutazione è stato ricavato un tempo medio approssimato, visualizzando dal grafico l'istante di tempo a partire dal quale il numero di veicoli resta costante, su un certo numero di run. Tale parametro è stato utilizzato come tempo di run per le successive verifiche, al fine di ridurre sia le tempistiche per l'analisi delle proprietà, e sia i consumi di memoria. La sintassi utilizzata è la seguente:

$$simulate\ N[<= bound]\ \{E1, \ldots, Ek\}$$

dove N indica il numero di simulazioni da effettuare, *bound* indica il tempo massimo per ogni simulazione, e $E_1 \ldots E_k$ sono le espressioni da monitorare e visualizzare. Effettuando i test su 10 simulazioni, il tempo ottimo di simulazione ricavato è stato di 650s.

Successivamente è stata costruita la query effettiva per la verifica della probabilità, utilizzando la seguenti sintassi di SMC Uppaal :

$$Pr[bound](\Psi)$$

Questa espressione stima, in maniera quantitiva, la probabilità che l'espressione Ψ risulti vera, fintanto che risulti vera l'espressione in $[\ldots]$ (in questo caso l'intervallo sul clock) [9].

L'ambiente di lavoro sul quale si sono effettuati i test, consiste in un PC desktop con CPU AMD FX8320 8 Core (4.0 GHz) a 64 bit, e 8 GB di RAM DDR3 a 1866 MHz.

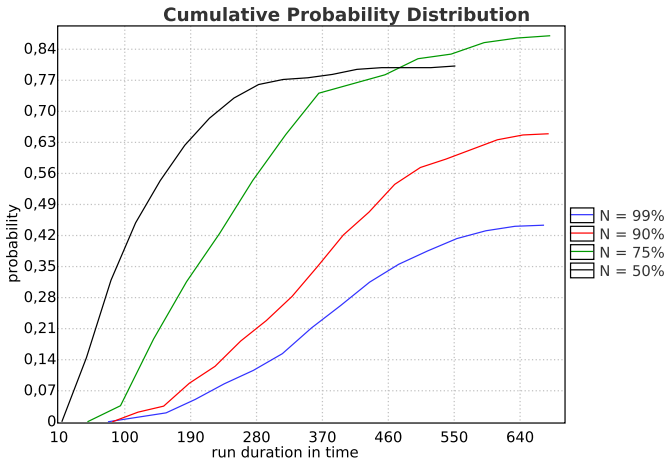
Veicoli (%)		Distanza (m) 100	300	500	700	1000
50	*Pr*	[0.7489, 0.8486]	[0.7907, 0.8905]	[0.4525, 0.5524]	[0.2773, 0.3772]	[0.0885, 0.1882]
	CPU (s)	66,979	61,113	196,434	222,287	123,976
	RAM (KB)	84.100	84.100	84.104	84.100	84.096
	runs	263	220	402	356	196
75	*Pr*	[0.8147, 0.9143]	[0.5674, 0.6674]	[0.0651, 0.1650]	[0.0004, 0.0989]	[0, 0.0973]
	CPU (s)	66,135	171,090	109,000	35,772	24,687
	RAM (KB)	69.696	69.696	69.692	69.696	69.692
	runs	193	380	167	54	/
90	*Pr*	[0.5982, 0.6981]	[0.1793, 0.2793]	[0.0004, 0.0973]	[0, 0.0989]	[0, 0.0973]
	CPU (s)	181,704	176,186	24,403	24,566	24,480
	RAM (KB)	69.696	69.692	69.692	69.696	69.688
	runs	368	287	54	/	/
99	*Pr*	[0.3937, 0.4937]	[0.0169, 0.1162]	[0, 0.0973]	[0, 0.0973]	[0, 0.0973]
	CPU (s)	224,830	64,315	24,698	24,845	25,403
	RAM (KB)	69.696	69.692	69.692	69.696	69.692
	runs	397	97	/	/	/

Tabella 1: Probabilità di Verifica Query, Tempi di esecuzione, Consumi di memoria

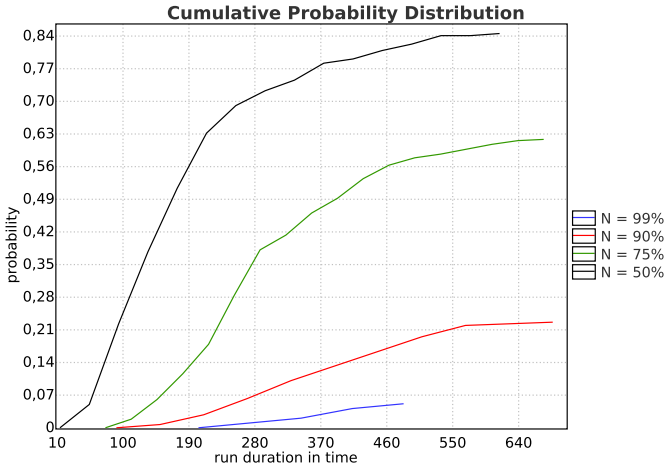
Per ogni percentuale di veicoli, in relazione alla distanza dal punto di anomalia, sono riportati i consumi di RAM, il tempo di CPU richiesto e l'intervallo di probabilità risultante. Si noti come i consumi di memoria RAM restino costanti entro certi valori, rispetto alla percentuale dei veicoli. In particolare, nel caso N = 50%, abbiamo un consumo di RAM leggermente superiore ($\sim$ 84 MB) rispetto agli altri casi ($\sim$ 69 MB). Per quanto riguarda il tempo di CPU richiesto, sono stati riscontrati valori compresi tra un minimo di 24 secondi, fino ad un massimo di 225 secondi (caso N = 99% e $d = 100m$). Infine sono stati riportati i valori dei *run* necessari alla stima dell'intervallo di probabilità, per ogni verifica della rispettiva proprietà.

In merito agli intervalli di probabilità, si nota dalla tabella come i valori siano inversamente proporzionali, sia al crescere della distanza che alla percentuale dei veicoli. Di seguito sono riportati i grafici relativi alle distribuzioni di probabilità. Da questi ultimi, si può osservare come in qualche grafico manchino i parametri indicanti la percentuale dei veicoli che riescono a ricevere un messaggio di anomalia.

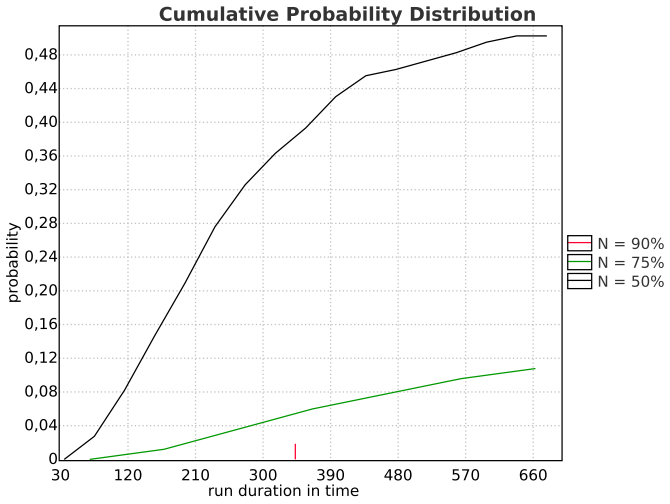
Questo accade poichè l'intervallo di probabilità è relativamente basso. Inoltre possiamo evidenziare come nei primi due casi ($d = 100m$ e $d = 300m$) il modello fornisce una buona stima di probabilità (circa l'80%), mentre nei restanti la probabilità oscilla tra il 10% ed il 48%.



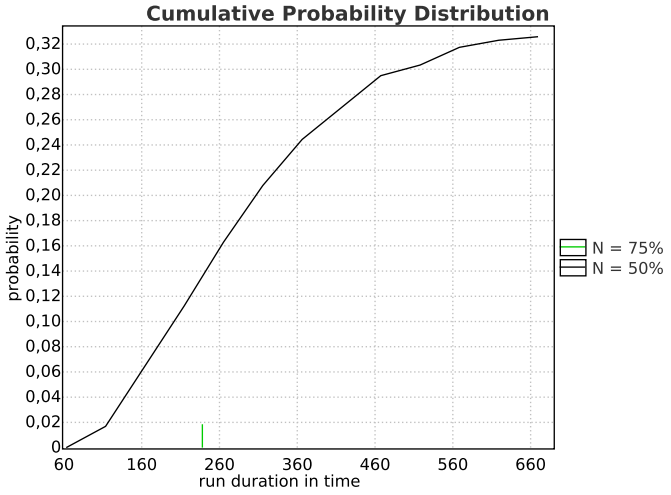
(a) $d = 100$



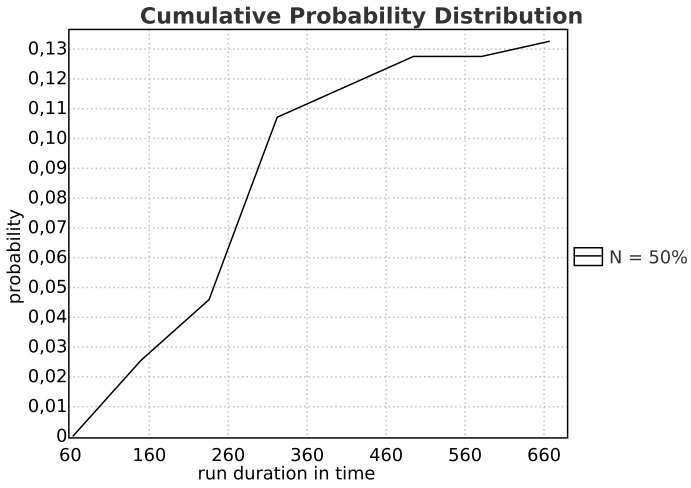
(b) $d = 300$



(c) $d = 500$



(d) $d = 700$



(e) $d = 1000$

Figura 6: Funzioni di distribuzione cumulativa relative ai set di Query effettuate

Riferimenti bibliografici

[1] Statistical Model-Checker: New SMC Extension of UPPAAL . http://people.cs.aau.dk/~adavid/smc/, 2015. [Online; Ultimo accesso 15 Maggio 2015].

[2] M.A. Berlin, Sheila Anand. Formal verification of safety message dissemination protocol for VANETs. *Journal of Computer Science 9*, 2013.

[3] Giorgio Basile. Studio e sperimentazione di un modello di disseminazione di annotazioni semantiche in reti veicolari. Master's thesis, Politecnico di Bari, 2012.

[4] Qiangyuan Yu, Geert Heijenk. Abiding Geocast for Warning Message Dissemination in Vehicular Ad Hoc Networks. *IEEE International Conference on Communications Workshops, 2008. ICC Workshops '08*, pages 400–404, 2008.

[5] Tamer Nadeem, Pravin Shankar, Liviu Iftode. A Comparative Study of Data Dissemination Models for VANETs. *Networking & Services, 2006 Third Annual International Conference on Mobile and Ubiquitous Systems*, pages 1–10, 2006.

[6] Gerd Behrmann, Alexandre David, Kim G. Larsen. A Tutorial on UPPAAL 4.0. *Department of Computer Science, Aalborg University, Denmark*, page 48, nov 2006.

[7] Camelia Avram, Adina Aştilean, Josè Machado. Modeling and Formal Analysis of Urban Road Traffic. *11TH International Conference of Numerical Analysis and Applied Mathematics*, 2013.

[8] Paolo Mazzoni. *Model Checking Tutorial*. Politecnico di Milano.

[9] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen. UPPAAL SMC Tutorial. *International Journal on Software Tools for Technology Transfer*, page 18, jan 2015.

[10] Floriano Scioscia. Introduzione al Model Checking. http://sisinflab.poliba.it/scioscia/teaching/lfc/LFC_17_Model_checking.pdf, 2015. [Online; Ultimo accesso 24 Maggio 2015].

[11] Uppsala University, Aalborg University. UPPAAL Official Site. http://www.uppaal.org/, 2015. [Online; Ultimo accesso 15 Maggio 2015].

[12] Peter Bulychev, Alexandre David, Kim Guldstrand Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, Zheng Wang. UPPAAL - SMC: Statistical Model Checking for Priced Timed Automata. *Electronic Proceedings in Theoretical Computer Science*, (85):1–16, 2012.