ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

ADAS ONTOLOGY

Elaborato del corso di: Web Semantico

Report di:
LORENZO CHIANA

ANNO ACCADEMICO 2019–2020

Indice

Capitolo 1

Panoramica

1.1 Cos'è un sistema ADAS

Il continuo progresso tecnologico ha portato, negli ultimi anni, le automobili moderne in veri e propri sistemi elettronici dotati di assistenza alla guida. Questi ausili elettronici sono indicati con l'acronico ADAS auto (Advanced Driver Assistance Systems) e, nonostante siano stati sviluppati principalmente per tutelare l'incolumità del guidatore e passeggero, con questa sigla si identificano anche tutti quei dispositivi presenti nell'auto per incrementare il comfort di guida.[1]

1.2 Sicurezza

Come già accennato l'obiettivo principale di un sistema ADAS è la sicurezza che si tramuta in riduzione del rischio di incidente grazie a diversi sistemi di controllo. Controlli che vanno dall'avviso di collisione a quelli di velocità.[1]

1.3 Perché le ontologie?

Gli attuali veicoli a guida autonoma in fase di sviluppo sono dotati di diversi sensori altamente sensibili come camera, stereo camera, Lidar, and Radar. Sebbene oggetti e corsie possano essere rilevati utilizzando questi sensori, i veicoli non possono comprendere il significato degli ambienti di guida senza

la rappresentazione della conoscenza dei dati. Pertanto un metodo di rappresentazione della conoscenza comprensibile da una macchina può essere una soluzione più che necessaria per colmare il divario tra gli ambienti di guida rilevati e l'elaborazione della conoscenza. Le ontologie vengono in aiuto in quanto sono le framework strutturali per la rappresentazione della conoscenza sul mondo o su una parte di esso, che è composto principalmente di concetti (classi) e relazioni (proprietà) tra essi.

1.4 Scopo progetto

Questo progetto si pone diversi obiettivi:

- studiare, comprendere ed estendere l'ontologia già fornita da Ichise Laboratory;
- raggiungere il seguente requisito: regolazione velocità al superamento dei limiti di velocità;
- utilizzare OWL API mediante linguaggio Java.

1.5 Analisi

1.5.1 Regolazione velocità per il rispetto del limite stradale

Con questo requisito si vuole andare a modellare uno scenario in cui lungo la carreggiata il veicolo debba modificare la propria andatura per il rispetto dei limiti di velocità imposti dal regolamento stradale. Come si potrà notare nel prossimo capitolo (2.1) un veicolo è in grado di modificare la propria andatura accelerando, decelerando o mantenendo l'andatura. Di conseguenza di vuole sfruttare ciò per modellare uno scenario dove un veicolo possa accelerare fino a una soglia massima che è il limite di velocità imposto, da lì in poi potrà solo procedere a velocità costante o decelerare.

Capitolo 2

Ontologie

In questo capitolo si andranno a mostrare le ontologie prese in esame: CarOnto, ControlOnto e MapOnto.

2.1 Classi di CarOnto

L'ontologia "CarOnto" contiene diverse classi e sottoclassi che modellano un veicolo, quali:

- ***CarParts***, ovvero le parti del veicolo, composte dal motore (*Engine*) e dai vari sensori (*Sensor*) come *Camera*, *CAN*, *GPS*, *Lidar* e *Sonar*.
- ***Trajectory***, ovvero la traiettoria del veicolo, composte dal tracciato (*Path*) e dai vari profili di velocità (*SpeedProfile*) come accelerazione (*Acceleration*), velocità costante (*CostantSpeed*) e decelerazione (*Deceleration*).
- ***Vehicle***, ovvero dal tipo di veicolo. Questa ontologia suddivide i veicoli in tre categorie:
  - *Automobile* nella quale sono presenti bus (*Bus*), veicoli regolari (*RegularVehicle*), veicoli speciali[1] (*SpecialVehicle*) e truck (*Truck*).
  - *Bicycle*;

[1] Con veicoli speciali l'ontologia intende quei veicoli che non sono soggetti alle consuete regole di precedenza come filobus o tram.

– *Motorcycle*;

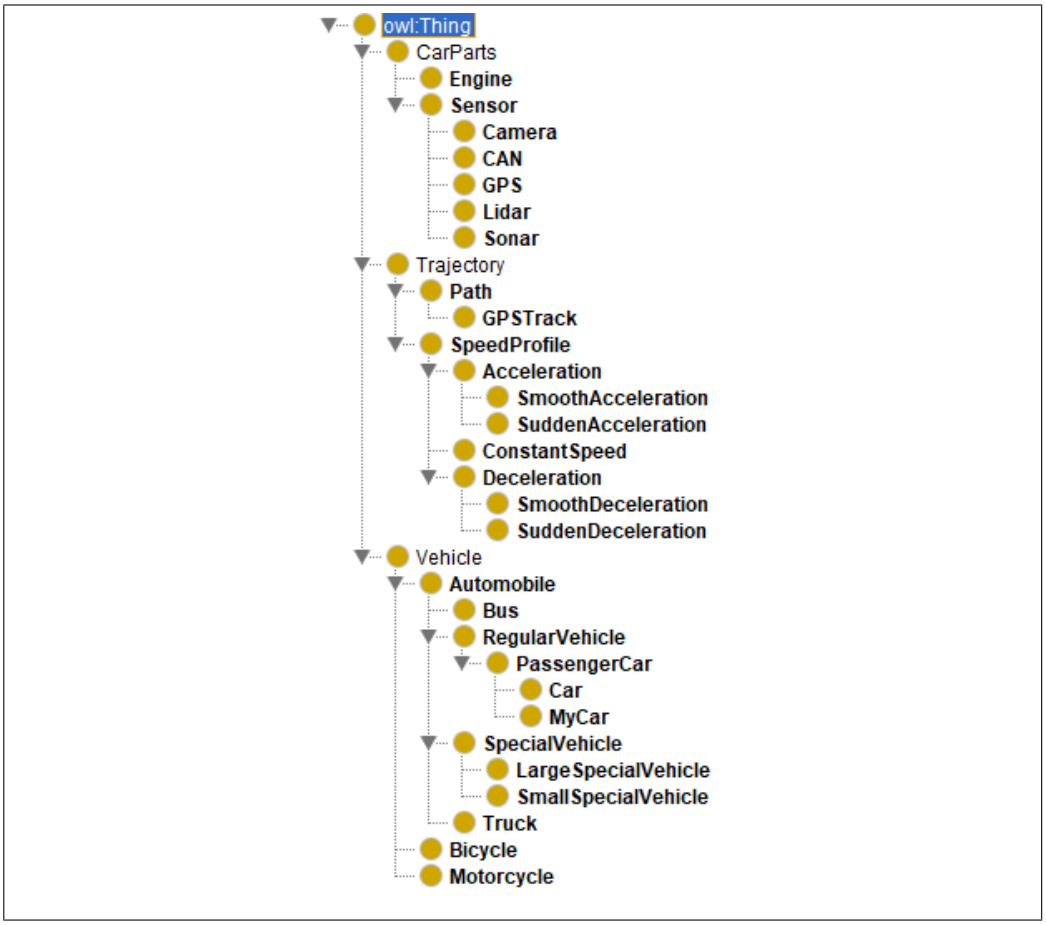


Figura 2.1: Classi di CarOnto

2.2 Classi di ControlOnto

L'ontologia "ControlOnto" contiene diverse classi e sottoclassi che modellano i vari controlli che vengono effettuati da un veicolo ADAS, quali:

- ***DrivingAction***, azioni effettuate dal veicolo come: partire (*Go*), proseguire (*GoForward*), far retromarcia (*GoBackward*), fermarsi (*Stop*) e così via;
- ***LaneChange***, modella il cambio di linea che può essere da destra (*RightLaneChange*) o da sinistra (*LeftLaneChange*);
- ***Path***, ovvero il percorso che il veicolo deve seguire;
- ***RoadCondition***, la condizione della strada;
- ***TrafficSignalControl***, modella le azioni da effettuare di in presenza di un semaforo[2] come: procedere in presenza del verde (*GreenGo*), fermasi in presenza del rosso (*RedStop*) e giallo (*Yellow*);
- ***Warning***, modella i possibili avvisi tra cui: quello di collisione (*CollisionWarning*), quello di deviazione dalla corsia (*LaneDepartureWarning*) e quello di superamento dei limiti di velocità (*OverSpeedWarning*).

[2]Traffic lights e traffic signal sono sinonimi di inglese e corrispondono alla parola "semaforo" in italiano.

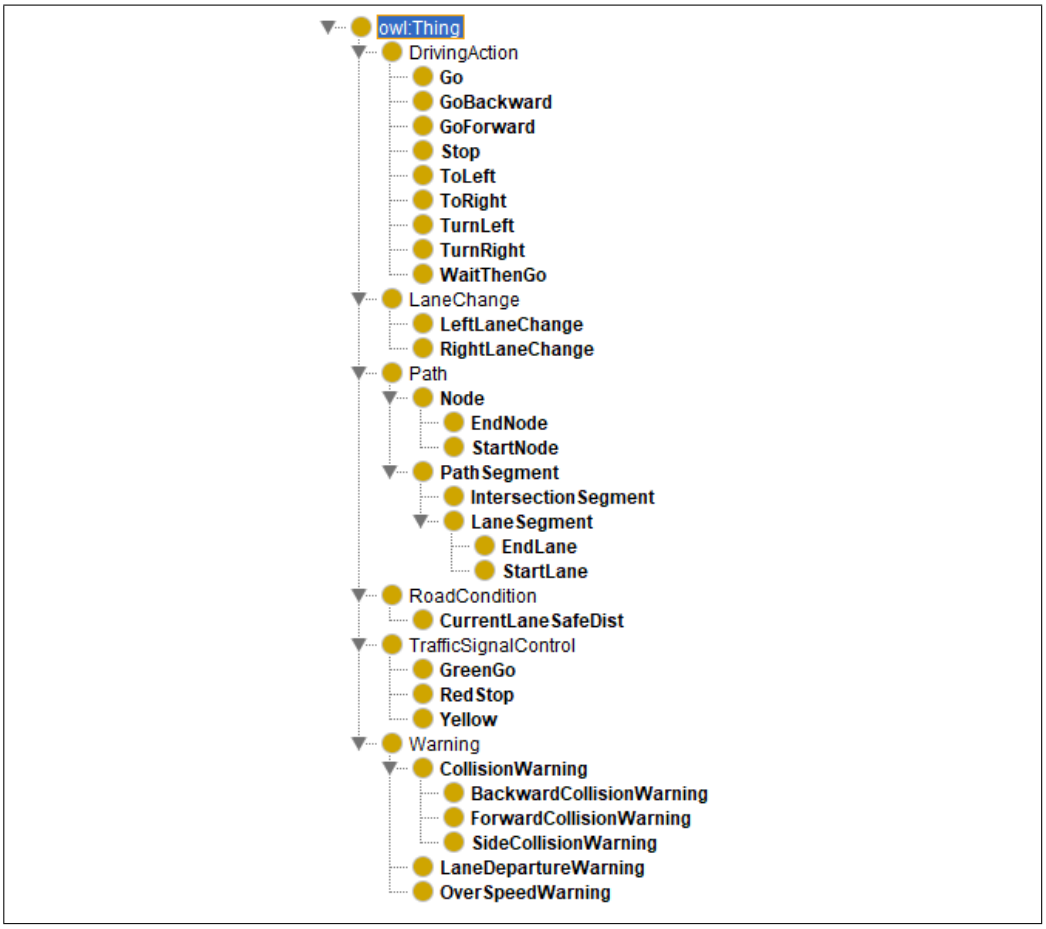


Figura 2.2: Classi di ControlOnto

2.3 Classi di MapOnto

L'ontologia "MapOnto" rappresenta la mappa stradale di un'area con segnaletica, quali:

- ***LivingThing***, possibili esseri viventi presenti nell'area come un umano (*Human*);
- ***Object***, possibili oggetti presenti nell'area come un palo di utilità (*UtilityPole*);
- ***Place***, il luogo che può essere:

  - un servizio (*Amenity*) come ristorante, negozio e così via;
  - un edificio (*Building*) come ospedale, banca, ufficio postale, ecc.;
  - un'infrastruttura (*Infrastructure*) come aeroporto, stazione, distributore di benzina o un casello;
  - un luogo naturale (*NaturalPlace*) come un lago, un fiume o una foresta.

- ***RouteOfTransportation***, la via di trasporto che contiene la parte e il tipo di strada (*RoadPart* e *RoadType*) e il limite di velocità vigente (*SpeedLimit*);

- ***Time***, il momento della giornata:

  - prima mattina (*EarlyMorning*);
  - mattina (*Morning*);
  - mezzogiorno (*Noon*);
  - sera (*Evening*);
  - mezzanotte (*Midnight*);
  - notte (*Night*).

- ***TrafficSign***[3], modella i vari segnali stradali presenti.

- ***TrafficSignal***, modella la presenza di un semaforo.

[3] "Traffic sign" è traducibile con "segnaletica stradale", da non confondersi con "traffic signal" che invece è "semaforo".[2]

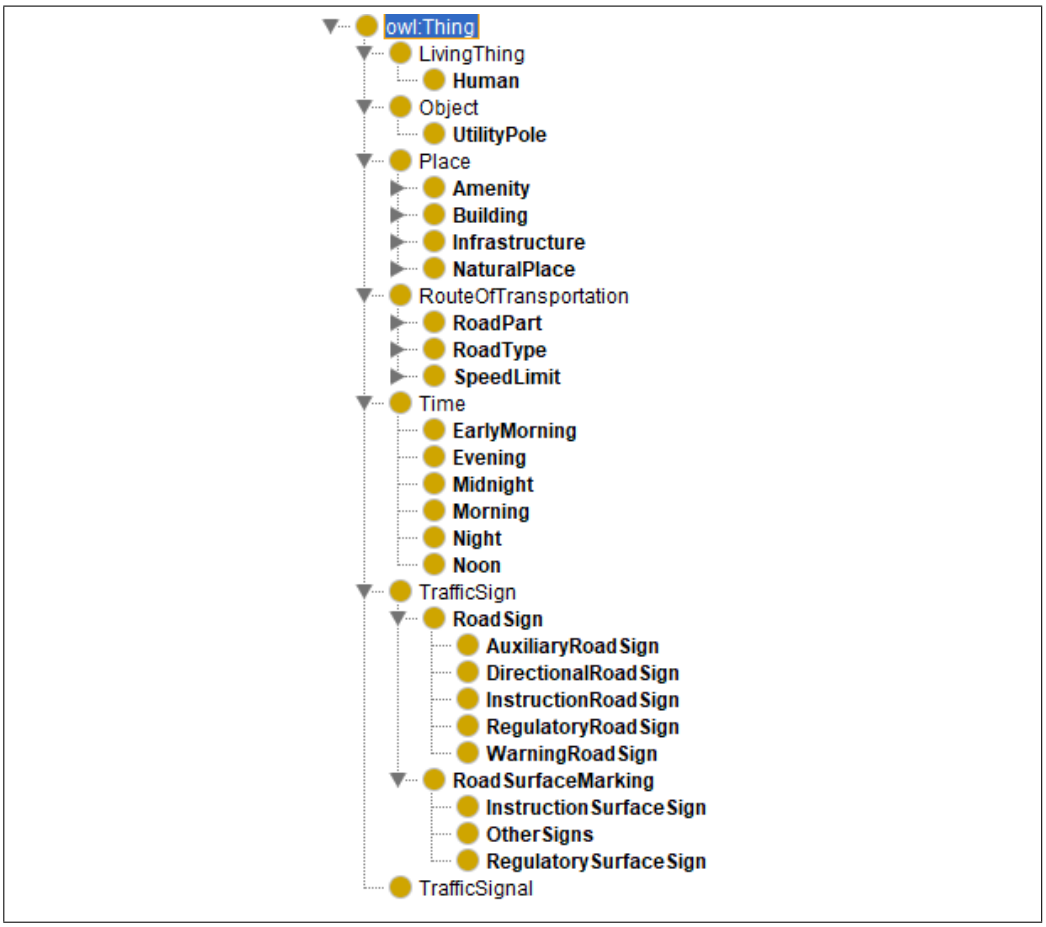


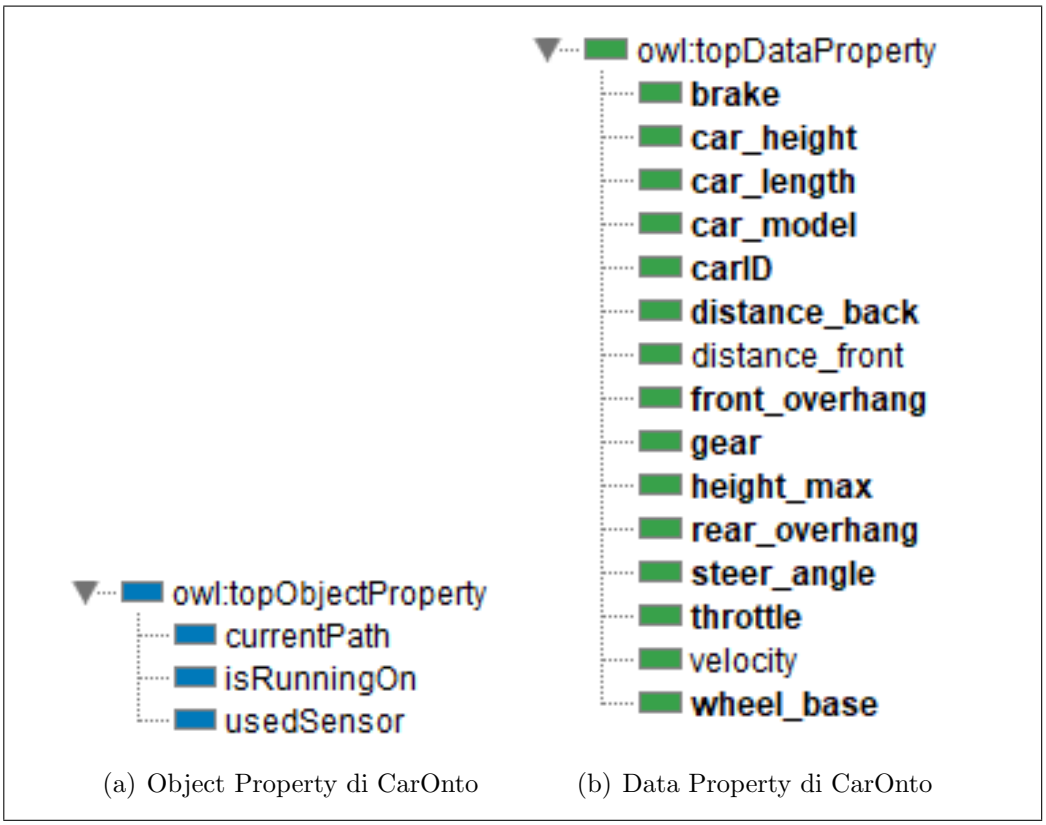
Figura 2.3: Classi di MapOnto

2.4 Proprietà

Le proprietà servono a descrivere le caratteristiche di determinate classi, creando quindi una relazione con altre classi o con valori veri e propri. Di seguito verranno mostrare le object property e le data property presenti nelle tre ontologie.

2.5 Classi riutilizzate

Di seguito esporrò le classi utilizzate nel progetto.



(a) Object Property di CarOnto (b) Data Property di CarOnto

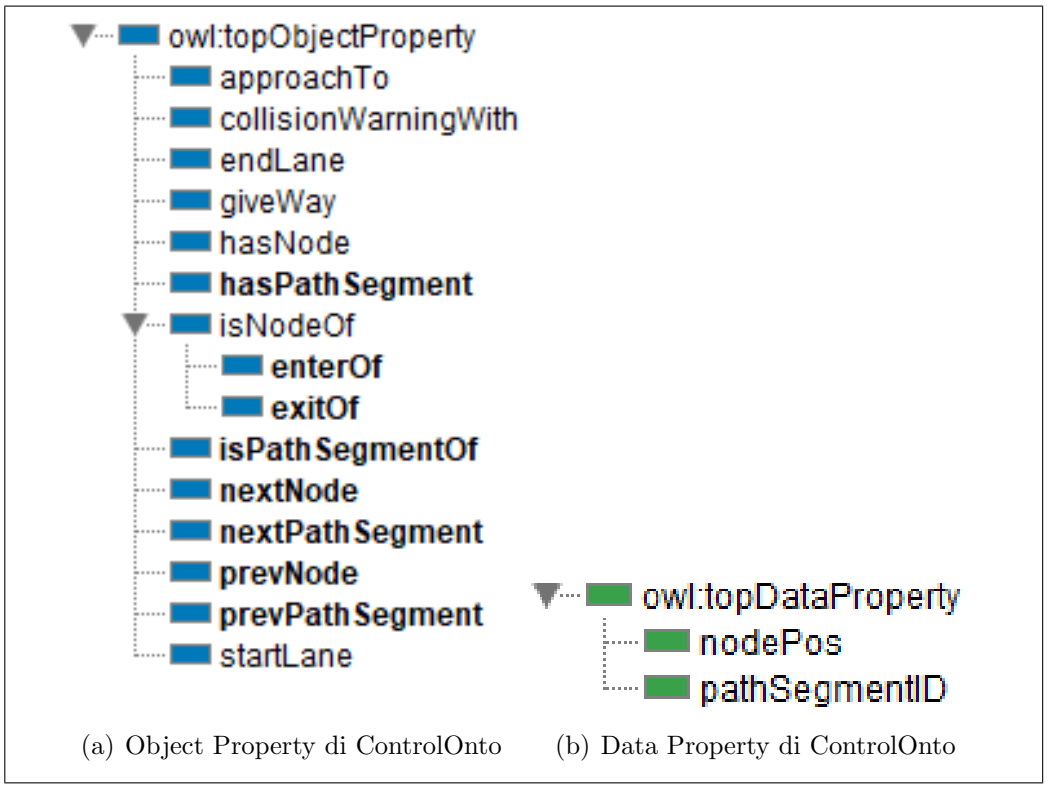
Figura 2.4: Object e data property di CarOnto

CarOnto

Per modellare il concetto di autoveicolo con sistemi ADAS si è utilizzata la classe *MyCar* anziché *Car*, entrambe sottoclassi di *PassengerCar*. Poiché *MyCar* è stata creata con l'intendo di dar la possibilità di modellare specifici modelli ai autoveicoli di qualsiasi casa automobilistica si è scelta questa classe anziché quella più generica per rendere il progetto riutilizzabile per un eventuale sviluppo futuro. Dato che vi è l'esigenza di gestire la velocità in relazione ai limiti imposti per legge nel progetto viene, inoltre, utilizzata la classe *SpeedProfile* con le relative sottoclassi *Acceleration*, *ConstantSpeed* e *Deceleration*.

ControlOnto

Per modellare e creare uno scenario dove l'auto percorre una determinato percorso su una carreggiata serviranno le classi *startLane* e *endLane* per



(a) Object Property di ControlOnto (b) Data Property di ControlOnto

Figura 2.5: Object e data property di ControlOnto

determinare dove iniziare e dove finire la simulazione. Inoltre, servirà la classe *overSpeedWarning* per modellare l'avviso di superamento dei limiti di velocità.

MapOnto

Per questo progetto serviranno: la classe *roadSegment* per modellare il percorso della simulazione; *oneWayLane* per modellare una corsia a senso unico; *speedLimit* per modellare il limite di velocità.

2.6 Proprietà utilizzate

Di seguito esporrò le proprietà utilizzate nel progetto.

2.6.1 CarOnto

L'object property *isRunningOn* è venuta in aiuto per sapere su che segmento di strada si trova l'autoveicolo. La data property *carID*, invece, è stata utilizzata per dare un identificativo al veicolo, utile per un eventuale sviluppo futuro del progetto nel quale saranno gestiti più veicoli alla volta.

2.6.2 ControlOnto

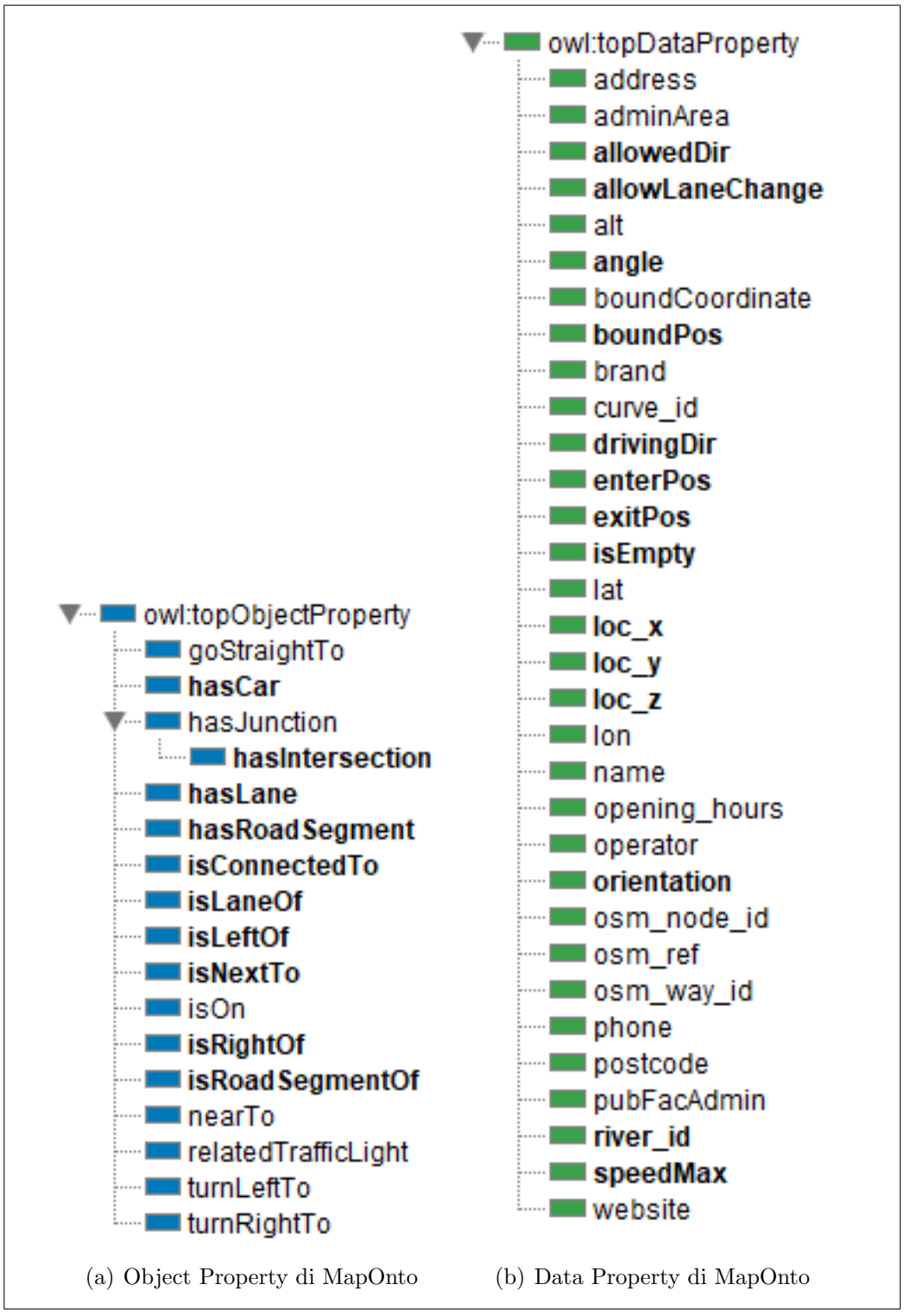
Dato che si vuole modellare la carreggiata come una sequenza di segmenti di strada (*PathSegment* classe madre di *LaneSegment*), si dovrà utilizzare l'object property *nextPathSegment* per creare la sequenza di segmenti. Inoltre, si avrà la necessità di assegnare un identificativo ad ognuno di questi segmenti, a ciò viene in aiuto la data property *pathSegmentID*.

2.6.3 MapOnto

Per la modellazione di una carreggiata a più corsie serviranno gli object properties *hasLane* e *isLaneOf*. Per definire il limite di velocità bisognerà usare la data property *speedMax*.

2.7 Estensione

Durante l'analisi delle ontologie precedentemente viste si è notato la mancanza di una proprietà importante nell'ontologia di controllo: *overSpeedWarningThan*. L'idea di questa nuova Object Property in ControlOnto è nata analizzando tale ontologia e notando che *CollisionWarning* è l'unica sottoclasse di *Warning* ad avere una proprietà che la descrive. Tale proprietà (*collisionWarningWith*) delinea con chi/che cosa il soggetto sta per collidere. Con *overSpeedWarningThan* si intende delineare quale limite di velocità si sta per violare. In ottica RDF sarà il predicato tra il veicolo (soggetto) e il limite di velocità (oggetto), si veda Figura 3.1.



(a) Object Property di MapOnto (b) Data Property di MapOnto

Figura 2.6: Object e data property di MapOnto

Capitolo 3

Tecnologie

In questo capitolo si vanno ad affrontare tutte le tecnologie utilizzate in questo progetto quali: RDF, RDFS, OWL, OWLAPI, SWRL, HermiT OWL Reasoner; dando anche una loro spiegazione.

3.1 RDF

Resource Description Framework o, abbreviato, RDF è lo strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati che permette di descrivere le relazioni fra le entità della parte di realtà che si vuole modellare. In questa tecnologia l'unità base per rappresentare l'informazione è lo *statement*, ossia una tripla del tipo *Soggetto - Predicato - Oggetto*. In questa tripla il soggetto e l'oggetto rappresentano una risorsa e il predicato la relazione tra esse. [3]

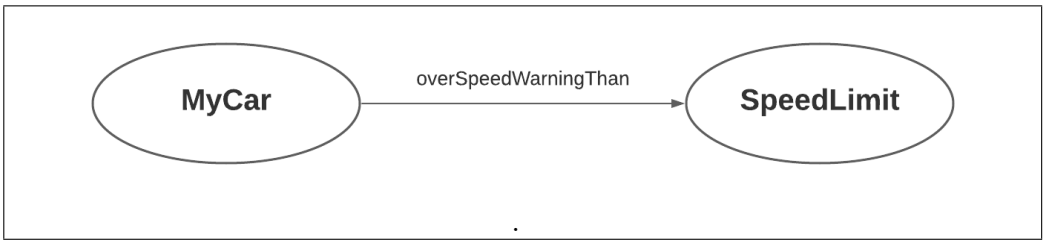


Figura 3.1: Tripla RDF con soggetto *MyCar*, predicato *overSpeedWarning-Than* e oggetto *SpeedLimit*

3.2 RDFS

RDFS, acronimo di RDF Schema, è una tecnologia nata dall'esigenza di dover definire vocabolari e semantiche, cosa che non è possibile fare solo con RDF. RDF Schema definisce un insieme di risorse RDF da usare per descrivere caratteristiche di altre risorse e proprietà RDF. [4]

3.3 OWL

OWL, acronimo di Web Ontology Language, è nato dall'esigenza di avere maggior potere espressivo rispetto a RDFS. I componenti principali di un'ontologia OWL sono tre: individui, proprietà e classi. Gli individui rappresentano gli oggetti nel dominio di interesse, le proprietà sono relazioni binarie (ovvero che collegano due oggetti per volta) tra individui, le classi sono gruppi di individui. [5]

3.4 OWLAPI

OWLAPI è un'API per Java 8 nata per la creazione, la manipolazione e la serializzazione di ontologie OWL. L'ultima versione si focalizza su OWL 2. Vi è una ampia documentazione con tanto di esempi pratici nel loro repository (`https://github.com/owlcs/owlapi/wiki/Documentation`).

3.5 SWRL

SWRL, acronico di Semantic Web Rule Language, è un linguaggio che consente la creazione di regole espresse in termini di concetti OWL (classi, proprietà, individuals). Ciò fornisce una capacità di ragionamento deduttivo più potente al solo utilizzo di OWL. SWRL supporta solo l'inferenza monotonica:

- le regole non possono essere utilizzate per modificare le informazioni esistenti in un'ontologia;
- le regole non possono ritirare o rimuovere informazioni da un'ontologia.

Una regola SWRL è composta da una parte antecedente (il corpo) e da una parte conseguente (la testa), entrambi costituiti da congiunzioni positive di atomi. Un atomo è espresso nella seguente forma: p(arg1, arg2, ..., argN) dove p è il simbolo del predicato e tra parentesi vi sono gli argomenti dell'espressione.[6]

Esempi applicati al progetto:
Mantenere una velocità costante

$$isRunningOn(?X,\ ?Lane) \wedge OneWayLane(?Lane) \wedge SpeedLimit(?Y) \wedge overSpeedWarningThan(?X,\ ?Y) \rightarrow constantSpeed(?X)$$

In questo caso il body è rappresentato da tutto ciò che è antecedente alla freccia e la head da ciò che ne viene dopo. Con questa regola il reasoner inferirà che se la head risulta vera allora anche il body lo sarà. Nello specifico se esiste una tripla che:

1. associa le variabili X e *Lane* con la proprietà *isRunningOn*;
2. *Lane* è istanza della classe *OneWayLane*;
3. Y è istanza della classe *SpeedLimit*;
4. associa X e Y alla proprietà *overSpeedWarningThan*;

allora X deve mantenere una velocità costante.
Accelerare

$$isRunningOn(?X,\ ?Lane) \wedge OneWayLane(?Lane) \rightarrow acceleration(?X)$$

In questo caso la tripla che deve esistere per far sì che X acceleri deve associare le variabili X e *Lane* con la proprietà *isRunningOn* e che *Lane* sia istanza di *OneWayLane*.

3.6 HermiT OWL Reasoner

HermiT è un reasoner per ontologie scritto in OWL. Data una ontologia OWL HermiT è in grado di determinare se essa sia coerente o meno, identificare le relazioni di inclusione tra le classi e molto altro. HermiT è il

primo reasoner OWL disponibile pubblicamente basato sul nuovo calcolo "hypertableau" che fornisce un ragionamento molto più efficiente di qualsiasi algoritmo precedentemente noto. [7] Per i motivi sopracitati e per la grande disponibilità di esempi trovati online anche, e soprattutto, dei vari esempi nella documentazione all'interno del Wiki del repository ufficiale di OWLAPI, la scelta del reasoner è ricaduta su HermiT.

Capitolo 4

Implementazione

4.1 Modellazione entità

Di seguito verrà esposto come le varie entità sono state modellate all'interno del progetto. Queste vanno a modellare e a inserire nell'ontologia i seguenti concetti:

- il mio veicolo con sistemi ADAS (MyCar);
- il percorso del veicolo (Map);
- i controlli sui vari segmenti (SegmentControl);
- il segmento di strada (RoadSegment);
- il profilo della velocità (SpeedProfile).

4.1.1 MyCar + MyCarUtils

Attraverso la classe singleton *MyCar* viene modellato il concetto di automobile con sistemi ADAS. È stata creata riutilizzando la classe ontologica *MyCar* di *CarOnto* e le proprietà *isRunningOn* e *carID* (vedi proprietà riutilizzate in CarOnto 2.6.1). Insieme alla classe *MyCar* è stata introdotta anche la relativa classe di utility *MyCarUtils* che permette, attraverso il metodo *addMyCar*, l'aggiunta di un nuovo individual della classe *myCar* all'interno dell'ontologia che rappresenta l'automobile con sistemi ADAS. Inoltre, attraverso il metodo *connectPropertiesToMyCar* si aggiunge il data

property carID al veicolo e col metodo *setMyCarPositionAndStartRunning* viene assegnata all'auto l'object property *isRunningOn* che specifica che è in movimento.

4.1.2 SegmentControl + SegmentControlUtils

Attraverso la classe singleton *SegmentControl* viene modellato il concetto di controlli sui vari segmenti di strada. Nello specifico, questi controlli sono inerenti a:

- corsia di inizio e fine percorso, tramite il riutilizzo delle classi *StartLane* ed *EndLane* di *ControlOnto*;
- direzione della corsa, tramite la classe *GoForward* dato che il percorso modellato non presenterà nessuna deviazione;
- avvertimento del superamento dei limiti di velocità, attraverso il riutilizzo della classe *OverSpeedWarning*.

Le proprità riutilizzate saranno *nextPathSegment* e *pathSegmentID* (vedi proprietà riutilizzate in ControlOnto 2.6.2), più la nuova object property *overSpeedWarningThan* (2.7)). Insieme alla classe *SegmentControl* è stata introdotta anche la relativa classe di utility *SegmentControlUtils* che permette la creazione di una sequenza di segmenti di strada attraverso il metodo *createPath* nella quale si va a stabilire qual è la corsia di inizio e quale quella di fine del percorso. Ad ogni segmento passatogli in input che andrà a comporre la rotta da percorrere verrà assegnato il data property *pathSegmentID* e create le varie triple *Segment_i - nextPathSegment - Segment_i+1*. Infine, verrà posizionata la vettura con sistemi ADAS sul primo segmento che andrà a comporre il percorso. Inoltre, in questa classe di utility sono presenti i metodi *addClassForSpeedAndGoForwardSWRLRules* e *connectPropertiesForSpeedSWRLRules* che aggiungono le classi e proprietà per le regole SWRL riguardanti i limiti di velocità e la direzione di guida.

4.1.3 Map + MapUtils

Attraverso la classe singleton *Map* viene modellato il concetto di percorso. Sono state riutilizzate le classi *RoadSegment*, *OneWayLane*, *SpeedLimit* e delle proprietà *hasLane*, *isConnectedTo*, *isLaneOf*, *goStraightTo*, *speedMax*

(si vedano le sezioni 2.5 2.6.3). Inoltre, tale classe contiene i due segmenti di strada di inizio e fine di tipo *RoadSegment*(sezione 4.1.4). Insieme alla classe *Map* è stata introdotta anche la relativa classe di utility *MapUtils* che permette, attraverso il metodo *addRoadSegment*, l'aggiunta due nuovi segmenti di strada: uno di inizio e uno di fine. Questi segmenti di strada vengono creati aggiungendo all'ontologia nuovi individual delle classi *RoadSegment* e *OneWayLane* per realizzare un segmento composto da due corsie a senso unico. Con il metodo *connectObjectPropertiesToRoadSegment* si va a definire:

- la proprità *hasLane* è inversa rispetto a *isLaneOf*;
- le asserzioni *RoadSegment - hasLane - LaneRight* e *RoadSegment - hasLane - LaneLeft*;
- l'asserzione *LaneRight of RoadSegmentStart - goStraightTo - LaneRight of RoadSegmentStop*.

Il metodo *addSpeedLimit* permette l'aggiunta nell'ontologia di un nuovo individual della classe SpeedLimit che rappresenta i limiti di velocità e, attraverso *connectSpeedMaxProperty*, si va ad assegnargli il data property *speedMax* settato a 50, modellando così un limite di velocità impostato a 50km/h.

4.1.4 RoadSegment

RoadSegment è una classe che modella un segmento di strada ed è composta da tre individual:

- road, il segmento;
- laneRight, la corsia di destra di cui il segmento è composto;
- laneLeft, la corsia di sinistra di cui il segmento è composto.

4.1.5 SpeedProfile + SpeedProfileUtils

Attraverso la classe singleton SpeedProfile viene modellato il concetto del profilo della velocità che un veicolo può sostenere, ovvero: accelerazione, decelerazione e velocità costante. Vengono riutilizzate le classi *Acceleration*,

ConstantSpeed e *Deceleration* di *CarOnto* (sezione 2.5). Insieme alla classe *SpeedProfile* è stata introdotta anche la relativa classe di utility *SpeedProfileUtils* che permette, attraverso il metodo *addSpeedLimit*, di settare le classi ontologiche dei tre profili di velocità.

4.2 Altre classi di utility

Per sviluppare questo progetto sono state create diverse classi di utility tra cui:

- ***OntologyUtils*** che contiene i diversi metodi per la creazione di un'ontologia OWL e delle sue parti come: la creazione di una nuova ontologia e, data un'ontologia, l'aggiunta di classi, individual, object property, data property, assiomi e così via;
- ***OWLOntologyUtils*** contiene i vari metodi per la gestione di una determinata ontologia OWL;
- ***SWRLUtils*** contiene i metodi per la creazione di regole, classi, object property e variabili SWRL;
- ***ReasonerUtils*** contiene il metodo per la creazione del reasoner HermiT e i metodi per ricavare la posizione di myCar e i nodi di inizio e fine;
- ***SimulationUtils*** contiene i metodi per la creazione della simulazione che verrà approfondita nella sezione seguente;
- ***IRIs*** contiene i vari IRI utilizzati all'interno del progetto;

4.3 La simulazione - SimulationUtils

La simulazione consiste nella creazione di una strada composta da due segmenti a due corsie a senso unico nella quale viene posizione un veicolo. Questo veicolo viene modellato in modo tale da non superare i limiti di velocità, impostati arbitrariamente a 50km/h, e deve procedere dal primo segmento al secondo mantenendo la corsia di destra. Attraverso il metodo

createSimulation di *SimulationUtils* viene creata tale simulazione inizializzando una nuova ontologia vuota con *createMyOntology* e i vari singleton attraverso il metodo privato *initClasses*. Mediante il metodo *createMyCarRoute* viene creato il percorso formato da due segmenti di strada. Tutti questi metodi fanno uso delle classi singleton e di utility esposte in dettaglio nella sezione 4.1 per la creazione della simulazione. I metodi *createSpeedLimitSWRLRule* e *createRunDirectionRule* vanno a creare le regole SWRL rispettivamente per la gestione dei limiti di velocità e della direzione di corsa che sono esposte in dettaglio nella sezione 4.3.1 e 4.3.3.

4.3.1 Regole SWRL per la gestione dei limiti stradali

All'interno del metodo *createSpeedLimitSWRLRule* vengono implementare le regole generali per il monitoraggio della velocità. Nello specifico vengono implementate le due regole portare in esempio nella sezione dedicata a SWRL (3.5), ovvero:

$$isRunningOn(?X,\ ?Lane) \wedge OneWayLane(?Lane) \wedge SpeedLimit(?Y) \wedge overSpeedWarningThan(?X,\ ?Y) \rightarrow constantSpeed(?X)$$

per la gestione del profilo di velocità quando viene segnalato l'avviso di eccesso dei limiti e

$$isRunningOn(?X,\ ?Lane) \wedge OneWayLane(?Lane) \rightarrow acceleration(?X)$$

invece nel caso in cui non venga segnalato l'eccesso dei limiti.

4.3.2 Test dei limiti stradali

4.3.3 Regole SWRL per la gestione della direzione

All'interno del metodo *createRunDirectionRule* viene implementata la regola generale per la gestione della direzione del veicolo che, in questo caso specifico, sarà quella di proseguire dritto. Tale regola è:

$$\text{isRunningOn(?X, ?Lane)} \wedge \text{goStraightTo(?Lane, ?NextLane)} \wedge \text{nextPathSegment(?Lane, ?NextLane)} \rightarrow \text{GoForward(?X)}$$

4.3.4 Test della direzione

4.4 Consistenza ontologia

Bibliografia

[1] automobile.it. Cosa sono i sistemi di sicurezza adas?

[2] Wikipedia. Vienna convention on road signs and signals.

[3] Antonella Carbonaro. Describing web resources in rdf.

[4] Antonella Carbonaro. Rdf schema.

[5] Antonella Carbonaro. Web ontology language: Owl.

[6] Roberto Reda. Rules for knowledge representation and reasoning in the semantic web.

[7] Oxford University. Hermit owl reasoner - the new kid on the owl block.