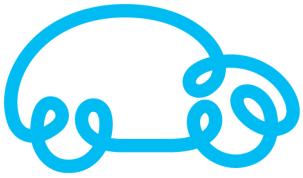
Dream-like simulation abilities for automated cars



DREAMS4CARS

Grant Agreement No. 731593

Deliverable:	D1.2 – System Architecture (Release 1)
Dissemination level:	PU – Public
Delivery date:	30/06/2017
Status:	Release 1, final

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731593

Deliverable Title	System Architecture (Release 1)			
WP number and title	WP1 Application domain requirements, Architecture and Product Quality Assurance			
Lead Editor	Sean Anderson, USFD			
Contributors	Mauro Da Lio, UNITN			
	Sebastian James, USFD			
	David Windridge, MU			
	Henrik Svensson, HIS			
	Rafael Math, DFKI			
	Mehemed Yuksel, DFKI			
	Elmar Berghofer, DFKI			
	Andrea Saroldi, CRF			
Creation Date	11/05/2017		**Version number**	3.6
Deliverable Due Date	30/06/2017		**Actual Delivery Date**	30/06/2017
Nature of deliverable	x	R - Report		
		DEM – Demonstrator, pilot, prototype, plan designs		
		DEC – Websites, patents filing, press&media actions		
		O – Other – Software, technical diagram		
Dissemination Level/ Audience	x	PU – Public, fully open		
		CO - Confidential, restricted under conditions set out in MGA		
		CI – Classified, information as referred to in Commission Decision 2001/844/EC		

Version	Date	Modified by	Comments
1.0	11/05/2017	Sean Anderson	Initial definition of contents
1.1	18/05/2017	Mauro Da Lio	Contribution to General Architecture and Agent Architecture
1.2	22/05/2017	Mauro Da Lio	Allocation of partners' sections and tasks
2.0	30/05/2017	Mauro Da Lio	Integration of contributions from DFKI and HIS
2.1	1/06/2017	Mauro Da Lio	Integration of contributions from USFD and MU
2.2	2/06/2017	Mauro Da Lio	Integration of USFD second contribution.

3.0	14/06/2017	Sean Anderson	Integration of DFKI, HIS and CRF contributions.
3.1	15/06/2017	Mauro Da Lio	Further contributions to sections 3.3.3-3.3.4 and section 6.3. First version of the Executive Summary.
3.2	16/06/2017	Mauro Da Lio	Further editing section 4. Moved section 4.8 to appendix.
3.3	16/06/2017	Mauro Da Lio	Further contributions to sections 3.3.4 and section 6.3.
3.4	19/06/2017	Sean Anderson	Check prior to review, minor edits.
3.5	27/06/2017	Sean Anderson	Edits based on reviewer comments
3.6	30/06/2017	Mauro Da Lio	Final check prior to submission

Definitions, acronyms and abbreviations

Abbreviation	Meaning
AD	Autonomous Driving
ADAS	Advanced Driver Assistance System
BG	Basal Ganglia (a number of subcortical nuclei of the brain)
DNN	Deep Neural Network
D2	Striatum D2-type receptors
ECU	Electronic Control Unit (a hardware device which allows implementing software in the car)
Euro NCAP	European New Car Assessment Programme
E/E	Electrical & Electronic
EGO	Autonomous vehicle
GPe	Globus Pallidus external (one of the nuclei of the basal ganglia)
EH	Electronic Horizon
GPS	Global Positioning System
GPU	Graphical Processing Unit
HIL	Hardware in the Loop
HMI	Human Machine Interface
HW	Hardware
LIDAR	Laser Imaging Detection and Ranging (a laser range sensor)
LWPR	Locally Weighted Projection Regression (a machine learning algorithm)
I/O	Input / Output
MIL	Model in the Loop
MSPRT	Multi-hypothesis Sequential Probability Ratio Test algorithm
NP	Normal Production
OBE	On Board Equipment
RV	Remote Vehicle
RTK	Real Time Kinematic
SAE	SAE International, formerly the Society of Automotive Engineers

SNr	Substantia Nigra pars reticulata
V2I	Vehicle to Infrastructure
V2V	Vehicle to Vehicle
V2X	Vehicle to any (where x equals either vehicle or infrastructure)
VRU	Vulnerable Road User
WTA	Winner Takes All algorithm

Executive Summary

This document is the first version of the System architecture.

The general architecture foresees three environments (Figure 1): the "wake state": the online real driving environment; the "dream state": the offline learning and optimization environment; and the "quality assurance" test environment.

This deliverable release (release 1) gives a detailed description of the architecture of the co-driver agent (online part) and general guidelines for the implementation of the dreams mechanism (offline part). The latter will be further specified in the next release 2.

The co-driver adopts a biologically inspired architecture with three main loops (Figure 2):

a) The "dorsal stream" which enacts hierarchical layered inverse models. It generates affordances from the sensory input.
b) An action selection mechanism ("basal ganglia") that operates on several levels of the hierarchy.
c) A "cerebellum" that learns forward models.

One important aspect of this architecture is the adoption of a topographic spatial representation of actions that is analogous to the human motor cortex. The dorsal stream creates affordances that result in active regions on this map. The dorsal stream also completely inhibits actions that have never to be selected, because they are related to probable collisions. This artificial motor cortex becomes the input for a statistically robust action selection mechanism bio-inspired by human "basal ganglia".

The agent has several potential benefits from this architecture:

1) The offline learning problem can be divided between learning separate loops (a, b, c listed above). For example, optimal inverse models for (a) can be synthesized with optimal control on learnt dynamics (c is learnt online). Section 7 gives an example of how this may happen.
2) The inhibition mechanism in the motor cortex will, in principle, *guarantee* system safety in the sense that, if the system has correctly learnt the inverse models, no dangerous action may ever be chosen (this form of proving safety may be of great help for market introduction). The agent architecture, and its inherently parallel topographic processing, can be implemented with Deep Neural Networks for which Graphical Processing Unit boards and middleware software (almost automotive grade) has become available.
3) The agent learning in the wake state consists of the learning of forward emulators (c); the offline (dream) learning consists of the learning and optimization of inverse models exploiting the forward models learnt in the wake state; the action selection mechanism can be exploited for action discovery (motivated learning).
4) The system, which now uses off-the-shelf automotive sensors is scalable to future perception system developmens. The DNN implementing the dorsal stream needs to be retrained for the new input signals of every training pair (the output activation of the motor cortex being the same in the same situation). This means that the collected scenarios used for dreams can be reused once a model of the new sensor system is available (Figure 2).
5) The system is portable to different vehicles because it may be retrained (in the same situations) for operation on vehicles with different dynamics (we will demonstrate this in the end of the project by porting the system to the CRF Jeep Renegade).

The hardware implementation of the agent foresees the use of an NVIDIA PX2 board, which provides two graphical processing units (GPU) to support deep neural networks used to implement the three loops.

The test sites and the test vehicles are described in section 5. There are three different tracks that can be used for the "wake" state driving. There are two "MIA" vehicles customized with sensors (including a Velodyne 32-HL Lidar) for development and one Jeep Renegade for final testing and comparison with the baseline agent of the AdaptIVe project.

As for what concerns the offline environment, this deliverable provides a description of the driving simulator environment, with an indication of several upgrades to be implemented, and a description of the main mechanisms that will be used to discover and optimize the agent behaviours. These include generalized motor babbling for bootstrapping the subsumption architecture that is part of the dorsal stream; optimal control, to synthetize optimal goal-directed actions; and bio-inspired exploratory learning. One example of the learning process is given in section 7.

Table of Contents

List of Diagrams

List of Tables

1 System Architecture

The Dreams4Cars system is composed of 3 different environments (Figure 1):

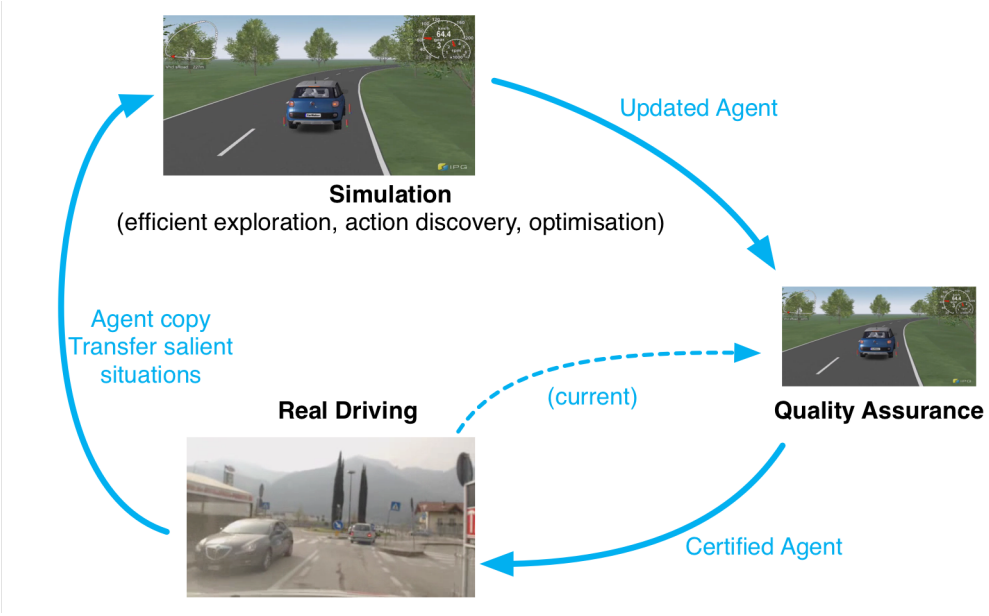


Figure 1: General system architecture.

1. The vehicle environment (Real Driving inset), in which the Co-driver[1] agent drives the vehicles (or observes the human driver). This is the "wake" state, in which the agent learns forward models and collects information about salient situations to be re-enacted in dreams.
2. The cloud simulation environment, implemented in the OpenDS simulator (Simulation inset). This is the "dream" state, in which a copy of the agent drives in a virtual world in situations that recombine (efficiently) the salient cases annotated at point 1. Optimization of the agent sensorimotor system and discovery of new actions takes place here.
3. The Quality Assurance environment (implemented in the Carmaker simulator) where a copy of the optimized agent (output from point 2) is tested against a growing library of test cases; the performance of the agent is assessed with several types of metrics (possibly including the Euro NCAP metrics); the progress in agent abilities is monitored.

The current release of this deliverable (D1.2, System Architecture, release 1) collects the work carried out in WP1.2 (system architecture of the runtime and offline components) and WP1.3 (specifications for the vehicles and cloud) and hence focuses on environments 1 and 2. Deliverable D1.4 will describe the Quality Assurance (WP1.4) and the third environment. An updated release of this same deliverable (D1.3 Architecture, release 2) is foreseen for month 24, which will include anticipations of the test cases library and of the Quality Assurance evaluation metrics.

[1] As already defined in D1.1, the agent driving the vehicles in this project is termed "Co-driver agent" or shortly "Co-driver", because it might share the control with the human diver when operating at different levels of automation. Hence, the terms "Co-driver" and "agent", together or in isolation, will be used in the document to refer to this artificial driving agent.

1.1 Current design practices and innovation of Dreams4Cars

The current design best practices (when simulators are used) are shown by the dashed arrow: the driving agent is copied in a simulator and tested in a number of scenarios. Evaluation of the simulated behaviour may lead to updates of the software that are carried out manually. Also, the test scenarios may have various origins, but in general they are mostly designed by the engineers that develop the system, based on their understanding of which important situations will be faced during real driving, or they are recorded situations.

Alternatively (see http://www.gputechconf.com), there are very recent examples of simulators that are used to generate synthetic data (input/output pairs), for training e.g., Deep Neural Network for object detection and classification.

Another very recent use of simulators (refer to the same http://www.gputechconf.com) is simulating sensors and perception system in various environmental situations to find conditions where mis-detection might happen.

Compared to the traditional design approach (even the most recent ones), the innovations introduced by Dreams4Cars may be summarized as follows:

a) Simulated scenarios are *generated from situations that the agent has seen* and annotated as salient (e.g., when something happens that was not foreseen by the co-driver). Any, even slightly, critical situations are likely to produce novel design scenarios that might be unknown to human designers (or the human designers might discover only with long analysis of recorded logs).
b) The agent can *automatically discover and optimize new strategies* (learning from its own "dreams") whereas in the traditional human driven approach, engineers have to diagnose code functionality, update the source code and test the new code (which is clearly slower, more erratic and error prone).

With point a) Dreams4Cars seeks to imagine new relevant situations from its own experiences; with point b) Dreams4Cars seeks to optimally update the agent abilities.

In order to implement point a) two elements are required: 1) the ability for the agent to form internal models or expectation of action outcomes, so that salient situations can be discovered; 2) efficient mechanisms to recombine recorded situations into effective dreams.

In order to implement point b) two other elements are necessary: 1) an agent that can self-reconfigure its perception-action architecture and 2) methods to discover and optimize behaviours.

2 Agent architecture

The agent architecture is bio-inspired: it reproduces the following loops of the human brain (Figure 2), which are described and motivated below.

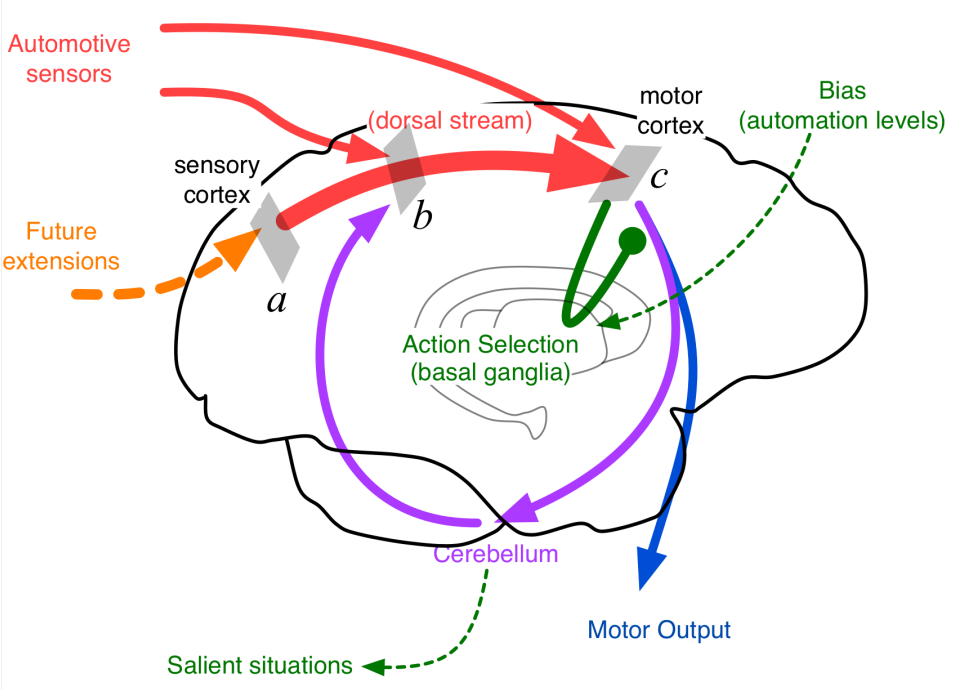


Figure 2: Agent sensorimotor architecture.

2.1 Dorsal stream – layered control

The "dorsal stream" takes sensory input ("sensory cortex", *a*) and gradually morphs ($a \rightarrow b \rightarrow c$) this input into a representation of possible actions (affordances), which are encoded as patterns of activity in the "motor cortex" (*c*). Overall the dorsal stream aims at *implementing simultaneous parallel action priming* according to the affordance competition hypothesis by Cisek [1].

Layered (subsumption) organization will be implemented[2] in the dorsal stream to produce strategies of higher-level of complexity (such as playing complex sequences of actions). Learning *and optimizing* these strategies (at all levels) is the primary goal of Dreams4Cars. This translates into learning the mapping from perception *to affordances*. Note that the goal of the dorsal stream is to produce *many potential actions*, not just one (the selection of the actual action to be executed will be carried out later). The fact that Dreams4Cars maps perception-to-(many)-affordances and then affordances-to-(one)-action in two steps, instead of perception-to-(one)-action in a single step, differentiates Dreams4Cars from other examples of learning perception-action maps such as, e.g., [2]. Separating affordance generation from action selection achieves the same advantages of the biological solution: 1) better adaptive behaviour, resulting from selection among a pool of potential actions and 2) the learning of new affordances allows the agent to develop new behaviours without needing to retrain the action selection mechanism (this translates to better scalability to complex situations).

In the co-driver dorsal stream, information is typically processed in two-dimensional topographic form. The sensory source may vary: it may be the visual field (here considered under the banner "future extensions"); or it may be an image representing the raw scans of a multi-beam LIDAR (or it may even be multi-sensory input

[2] The details of the implementation are given in the next sections.

such as cameras and LIDAR); or it may be a bird's eye view of the surrounding environment with obstacles, such as can be derived from the high-level output of many automotive sensors. For example, digital maps combined with localization and sensors that provide object detection and classification may be sufficient to produce a bird's eye view of the scene (the various arrows indicated under the banner "Automotive sensors" stand for current technologies that will be used in Dreams4Cars). Direct use of high-level automotive sensor output to compute the activation patterns in the motor cortex is also possible (and indeed, this is what is implemented in AdaptIVe).

Hence, once perceptual data are mapped into activity patterns representing possible actions in the motor cortex, the Dreams4Cars system becomes largely independent from the actual perception system and sensors and hence more *portable* between different vehicle systems (portability/interoperability is one point of WP1.2).

Whichever the sensor system, the final result of the dorsal stream processing will thus be a pattern of activations in a two-dimensional map that we call "motor cortex" in analogy with the biological ones. Each active region in the "motor cortex" will represent a combination of lateral and longitudinal control that corresponds to a distinct feasible action (this approach was first used in InteractIVe, [3], Fig.7).

In this way, affordances are represented by active regions (typically humps) of the cortex and learning new motor strategies means updating the dorsal stream so that new affordances appears as new active regions. The height of the motor cortex humps will encode the merits of each action (inversely related to its cost) or their *saliency* from the point of view of the agent affordable actions.

Actions leading to collisions (D1.1 section 2.4 system requirements priority, point 1-a Safety) will be *completely* inhibited so they will never be chosen. In principle, we will therefore have a collision-free system (as long as there is at least one feasible action, the system has learnt correct mapping and all relevant objects have been perceived). A system that "knows" more strategies will generate more affordances and will have more and better options. A well-trained system will in addition evaluate the merit of each strategy more precisely and hence, for example, might be able to adopt cautious behaviour when required (such as situations where some part of the surrounding environment is precluded from the sensors). Interestingly the system should be able to discover by itself some rules of the driving code (e.g. moderating speed in certain circumstances).

Actions violating priorities of lesser importance (D1.1 section 2.4 system requirements priority, points 1-b Traffic rules etc.) will be only partially inhibited, so that the system will choose to break a traffic rule if this is necessary to avoid a collision. For example, let us imagine that a vehicle is approaching an intersection and, because of another vehicle's fault, there is no collision free trajectory except for one violating the speed limit (accelerating for passing before the oncoming obstacles). In this case, partial inhibition for traffic rules allows the formation of a shallow peak in the motor cortex corresponding to the strategy that violates the speed limit and the agent will choose this option if there are no other choices (many examples of this type could be made).

2.2 Basal Ganglia - action selection

The action-selection mechanism may operate at several levels within the hierarchical subsumption architecture. At the lowest level, it operates on the "motor cortex": it takes the motor cortex as input and returns a copy of the motor cortex with only one active hump (which is the action that is gated to the motor system). The selection occurs by suppressing all the affordance humps except one, which is represented by the green round-arrowhead line in Figure 2. At a higher level, it may prove to be necessary to model a separate basal ganglia loop on the output of the DNN whose output is grounded symbolic representations of salient environmental features and potential actions. Thus, using basal ganglia circuitry, each level of the subsumption architecture would be "cleaned up" before passing its output to the level below.

In biological systems action selection is carried out by the basal ganglia (BG): a number of sub-cortical nuclei that cooperatively achieve this task. The selected action is not simply the highest hump. In other words, the selection mechanism is not Winner Takes All (WTA); rather it is speculated that the BG implement a more sophisticated algorithm such as MSPRT (multi-hypothesis sequential probability ratio test) [4], which may be seen as an algorithm to carry out *optimal decision making between alternative actions with time and error rate constraints*.

With MSPRT action selection becomes robust to sensory noise (it integrates evidence for actions over some time) and robust to control noise (e.g. a slightly lower but wider hump may be preferred to a higher but narrower one which would require very accurate control). Also, MSPRT provides some *persistency* in selected action, such that the agent does not continuously change its mind. With this respect, it is worth commenting that in the current AdaptIVe co-driver, which is based on WTA, some hysteresis had to be included in the selection mechanism. Indeed, it happened that in an overtake situation, with a very small speed difference, car-following and lane-change options were almost equivalent, and the agent would oscillate between the two because of noise.

The BG circuitry allows many selection mechanisms to work together. Moreover, it has not only "selection circuits" for allowing actions to occur, but also a NO-GO circuit which can actively prevent selection (The D2-pathway via GPe→ SNr). This could be useful within this project which has to actively suppress certain subsets of actions (avoiding collision for example).

The BG is also endowed with a mechanism - neuro-modulatory control via dopamine - for changing its 'set-point' for selection in terms of ease of selection. This is useful in learning, which can benefit from 'promiscuous selection' when exploring the environment, and more conservative selection when mature.

Finally, the selection mechanism may be biased (green dashed arrow) to implement high-level directives, such as changing the mode of operation at different levels of automation.

Thus, in Dreams4Cars the action-selection mechanism will be inspired by the above principles, either being functionally equivalent or by reproducing the various nuclei and processing that occur there (this can also be implemented with topographic computation).

2.3 Cerebellum – forward models

Dreams4Cars will implement forward models. Generally speaking, forward models should produce an anticipation of the *entire* sensory input, such as, e.g., predicting the next bird eye's view configuration.

We may break down this function into the prediction of the host vehicle trajectory, which depends on the action that is selected (or potentially selected) by the agent, and the prediction of the other agents' behaviours.

Dreams4Cars will evaluate two approaches.

1) Hybrid analytical-learning approaches. One approach is learning the parameters of a vehicle model (e.g. a bicycle vehicle model) and supplementing it with learning the un-modelled aspects with a general learning framework such as LWPR. This approach is expected to be more robust than learning the entire dynamics, because the vehicle model introduces some a-priori knowledge constraining the learning process. However, one important reason for a hybrid approach such as the above, using LWPR for the learnt part, is that symbolic derivatives for the forward dynamics are available (both the equations of the model and LWPR allows symbolic differentiation), which in turn allows efficient formulation of indirect Optimal Control problems (which is using a Variational formulation to generate the co-state equations). This way we should have a robust and efficient mechanism to synthetize optimal behaviours [5] from learnt dynamics (WP3.4, WP3.5).
   To predict the behaviour of other vehicles, a simplified mirroring mechanism may be used, which may exploit the forward models built as described above. Simple mirroring mechanisms have already been used in AdaptIVe and InteractIVe projects [3].
2) Neural Networks. Neural Networks (inspired by the adaptive filter model of cerebellar function) and/or Deep Neural Networks may be used for either learning the un-modelled dynamics (replacing LWPR) or for learning the complete dynamics (which may be particularly useful for learning those agents that have no simple/immediate mathematical model). The advantage of this second option is that it will be more consistent with the DNN implementations of the layered control and action selection loops (hence using same training tools, see below). Also, in this case we may exploit some ideas related to inversion of DNN such as to generate the expected input from patterns of activation representing various output symbols [6], [7] . The advantage of this approach, besides being more general, is that "interpolation" between symbols becomes possible and generates interpolated input: in [7] hy-

brid/imaginary chairs, cars, tables are produced in this way. The mechanism should however be extended to generation of interpolated behaviours and situations (this will be a research topic).

In biological systems forward models may have several uses: a) in overt action (online use) they may be used to produce expectations of the sensory feedback, and to enhance and process sensory information; b) in covert actions (offline use) they may be used for motor imagery and various forms of simulation of actions.

Dreams4Cars foresees 3 particular uses:

1) A self-monitoring system (online use) to detect anomalies in vehicle dynamics or sensor data quality and to detect failures. The implementation of this subsystem is described in section 3.3.6.
2) Detection of novelties (online use), i.e., mismatch between agent prediction and what happens at mostly the higher-levels of the sensorimotor architecture. This is the mechanism to annotate salient situations for future dreams. The implementation of this subsystem is described in section 3.3.5.
3) Learning a model of the world for offline simulations (dreams).

Concerning the offline use the forward models learnt by the agent in the wake state may either be used directly (in the case of optimal control) or indirectly in the OpenDS simulator. In the latter case the models learnt online will work as a "blue-print" for the forward dynamics models of the simulator environment. With this respect, we should indeed note that dreams carried out in the simulator environment (Figure 1) are not direct interactions of the agent with its internal models (as it is for human dreams) but rather interactions of the agent with an "equivalent" copy of its internal model that runs in the simulator (it is more like a human interacting with a driving simulator tuned to what he has learnt, rather than dreaming directly in his own mind). This indirect use may be justified because the cloud environment offers many advantages, such as collecting salient situations from multiple agents, (potentially) more computational power as well as a business model for the deployment of the Dreams4Cars technology.

3 Hardware and Software Implementation

3.1 Organization of the development process (from AdaptIVe to Dreams4Cars)

The AdaptIVe co-driver is implemented in an in-vehicle Linux Debian computer system (VBOX 3600, based on an Intel i7 CPU).

For fast prototyping and testing the software is developed in Wolfram's Mathematica. The MathCode add-on is used to produce C++ source code which can be compiled for model-in-the-loop (MIL) (CarMaker/Matlab) and hardware-in-the-loop (HIL) or demonstrator vehicle (VBOX). For the analysis of the logged data and software development/testing/debugging the MathCode add-on allows calling both the compiled functions (i.e., exactly the same used in MIL/HIL environment) as well as replacing them individually with the interpreted source Wolfram Language version (which can be quickly edited and tested). If necessary, any other C++ and Java source code may be wrapped as an external MathCode function and used within the Mathematica prototyping environment. This allows expansion of the current development environment with new tools and libraries required by Dreams4Cars (for example from contributing partners) and enables a smooth transition from the AdaptIVe system to Dreams4Cars.

The interfaces (input and output signals from the co-driver module are defined in a master spreadsheet document. Software scripts in Ruby parse the master interface document and generate: a) the Mathematica input/output functions, b) buses for Simulink to be used in the CarMaker MIL environment, c) header files for C++ code (this has greatly reduced interface errors) for HIL implementations. We plan to maintain this organization for the interfaces of Dreams4Cars.

3.2 Current co-driver architecture (AdaptIVe)

From the point of view of the software, the AdaptIVe co-driver implements a *two-layer* control architecture (dorsal stream, Figure 2).

The bottommost layer is made of Optimal Control motor primitives: these are produced with a third-order linear kinematic plant that allows the primitives to be produced in closed form, which in turn allows fast enough computation of the motor cortex activation map. The caveat is that, by using a linear kinematic plant model, simplifications and limitations are introduced: the model is good for smooth vehicle control, such as in ordinary driving conditions, but neglects non-linear phenomena that may happen when it is necessary to drive closer to the limit of manoeuvrability of the vehicle, such as in critical situations.

The second layer of the current co-driver evaluates three strategies: a) change to the left lane (if a lane is available), b) remain in the current lane c) change to the right lane.

With the above two layers, the system is capable of following one road path, such as given by a navigator, and adapting to speed limits, road curvature, surrounding vehicle traffic (including changing lane if the vehicle in front is slower), stop and start in congested traffic, emergency stop (or evasive manoeuvre) for obstacles in the lane and stop and start at traffic light that support I2V (essentially the system implements level 3 automation in roads where intersections –including pedestrian intersections– are all regulated by intelligent infrastructures with I2V).

To make one example, Figure 3 shows one of the many situations tested in AdaptIVe, which is the case when the host vehicle is being overtaken by another vehicle (Figure 3, left).

Figure 3, right, shows regions of the motor cortex that are completely inhibited (red) or partially inhibited (yellow) that respectively correspond to colliding or moving too close to the overtaking vehicle (see Figure 3 caption for more details).

The actual motor cortex activation can be imagined as a surface over the area depicted in Figure 3, right. In AdaptIVe there actually are 3 of these surfaces, one for each of the 3 level-2 strategies mentioned above.

The current action selection mechanism works as follows: 1) for each of the three level-2 strategies, an optimal action is selected using the WTA criterion, 2) then the three, discrete, optimal actions are compared using a

modified WTA criterion that allows introducing biases. For example, a bias in favour of the right lane is introduced to make the agent change to the right lane whenever it is not inhibited.

In other words, the highest peak of the three surfaces is selected (WTA) and of the three peaks one is selected with the modified WTA criterion.

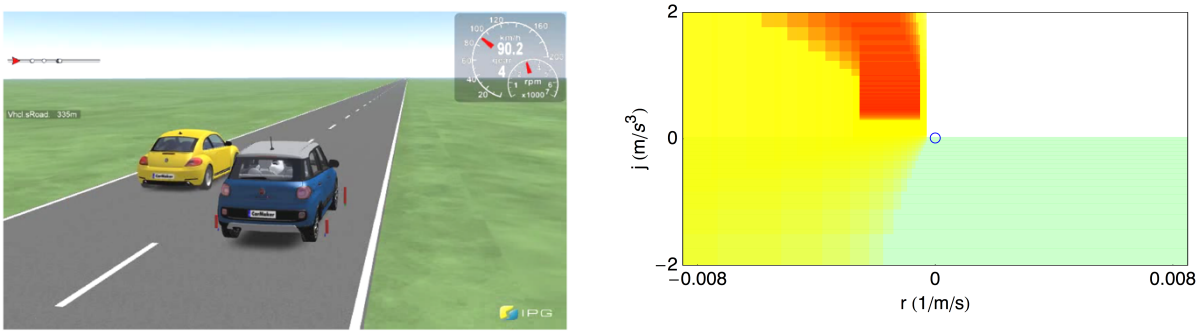


Figure 3: Left: the host vehicle (blue) is being overtaken by another vehicle (yellow). Right: the motor cortex activation as computed by the AdaptIVe co-driver. The *x* axis is the steering rate, the *y* axis is the longitudinal jerk (the acceleration rate). Every point on the right map represents a possible instantaneous action of the co-driver (the control space has dimension 2). The red region represents complete inhibition (by steering left and increasing the acceleration the host vehicle would collide with the yellow car). Note that the inhibited region is computed as the union of many rectangles, which represent estimated future positions (at discrete times) of the yellow car, according to host vehicle simple motor imagery. The yellow region represents actions that produce trajectories that would end too close to the overtaking car. The green area stands for actions that do not violate the speed limit. The small circle is the selected action (lane keeping at constant speed).

3.3 Implementation of the Dreams4Car architecture

Having clarified how the baseline AdaptIVe architecture works, we are now in a position to introduce the future implementation of Dreams4Cars.

3.3.1 Single motor cortex

One important change in the architecture is the use of only one motor cortex: where the AdaptIVe system computes three different activation maps for the three higher-level strategies, this option is not practical for a system that will have to develop an increasing number of different strategies with several layers of control; that would also call for developing corresponding levels of selection.

Instead, the solution will adopt a single representation of the motor space and have various peaks representing the different affordances appearing on the same map. Biases (such as favouring moving to the rightmost lane whenever possible) may be introduced in a twofold way: a) by providing more strength to the preferred action or b) by providing a separate salience map as a secondary input to a BG-like selection mechanism.

Learning new affordances means that the system will recognize the situation and produce a new activity peak for the combined longitudinal-lateral control that initiates that action. Also, the height of the activation in every point of the action space will encode the merit (inverse of difficulty) of controlling the system so that the width of an activity hump will also tell how precise the control must be for that particular affordance (which in turn will favour selection of larger humps, hence more robust actions).

3.3.2 Deep Neural Network implementation and GPU computing

While the software for AdaptIVe is coded by an engineer, Dreams4Cars needs the ability to re-configure its own sensorimotor system as learning new strategies progresses (project objective O1).

Quite recently, Deep Neural Networks (DNN) have received increasing attention because of the increasing availability of efficient software, but, above all, because dedicated GPU computing units greatly accelerate both the inference and the training.

In particular NVIDIA has recently made available to selected developers a new computing unit (DRIVE PX2) with two PASCAL GPUs which delivers more than 20 TOPS. This board has been adopted in several demonstration projects, including an end-to-end DNN trained for lane following in generic roads even without lane marking [2], [8]. Many applications, mostly concerning classification of visual scenes, running in real time, have been showcased at the latest GPU Technology Conference 2017 (http://www.gputechconf.com).

The unit is supplied with low-level and mid-level software. The low-level software includes CUDA drivers that can run 8-bit inference (4 times faster than 32-bit integers) and TensorRT, a software that optimizes a trained DNN for even faster inference time. At higher level, a software suite called “Driveworks” provides Application Program Interfaces for sensor input, vehicle control output, DNN management, tools for sensor calibration and for data logging and example of working pre-trained networks.

While the NVIDIA solutions are primarily geared towards perception, and in particular visual perception, they also appear to be well suited to supporting the execution of the neural networks that can be used to run the architecture of Figure 1 (because, essentially the networks of Figure 1 operate on tensors, e.g. topographic representation of actions); higher-levels of the dorsal stream might be implemented with more traditional neural networks, which are also supported.

The hardware implementation of the architecture of the agent will therefore be an NVIDIA PX2. Several DNN software suites are supported, there is not yet a commitment towards a specific package.

3.3.3 Interfaces

The co-driver interfaces include the sensor signals, the motor output signals and the annotations of salient events (Figure 1).

In the initial implementation, we will adopt the exact same interface signals that are in use for the project AdaptIVe. There are currently about 500 signals that can be divided in different categories (see Tables 1 and 2, taken from D1.1, tables 4 and 5).

Table 1: Current co-driver input (AdaptIVE).

Categories	Description
Vehicle Data	List of information derived from on-board vehicle sensors: speed, lateral and longitudinal acceleration, yaw rate, steering wheel angle, steering wheel speed, etc.
Localisation	Localisation of the vehicle, as resulting from combination of data about absolute position of the vehicle based on the global positioning system and vehicle motion sensor information (yaw rate, acceleration and vehicle speed).
Automation level	The current automation level is used by Co-driver to change its way of operation.
Fused object data	List of objects with their relative position and velocity in front of, behind, at the left and right side of the ego-vehicle.
Road Description data	Road path described as resulting from Electronic Horizon, includes road geometry and other information about the road.
V2X data	V2X data are all messages received from communication with other road users, infrastructure or cloud.
Requested driving style	Some parameters about the driving style selected by the driver. These parameters include the selected target speed.

Table 2: Current co-driver output (AdaptIVe)

Categories	Description
First Trajectory	Set of parameters defining the first trajectory (space-time).
Second Trajectory	Set of parameters defining the first trajectory.
Internal state parameters	Extra parameters used for testing

The workflow for the management of the interfaces has been developed and refined in the AdaptIVE project and consists of a centralized repository, where new signals and attributes can be defined (or removed) and of a number of procedures and scripts that generate the files for the various software environments (header files, Matlab buses, Mathematica functions accessing the data etc.). These procedures allow to maintain the consistency of the various environments interfaces (vehicle, simulators, etc).

For Dreams4Cars, the interface specification will evolve in the project but we still plan on having automatic generation of the files that specify the interfaces in the various environments.

For Dreams4Cars, extensions of the interface signals is planned. The main additions fall in three categories:

1) Maps. The digital maps for Dreams4Cars will have to be enhanced in several aspects (see section 3.3.4). Hence the interfaces will have to be changed.
2) Semantic annotations will have to be included in Table 2.

3) Raw sensor data. Although Dreams4Cars is not about improving sensor technology, it has to be open to the use of new perception systems that might be developed in the future. The project will thus evaluate possible expansions shown in Figure 2 and in particular inclusion of lower-level sensor data (or even raw sensor data).

3.3.4 Digital Maps

ADASIS is a standardization initiative for a data model to represent map data ahead of the vehicle (the so-called ADAS horizon). This initiative is coordinated by the ADASIS Forum, involving vehicle manufacturers, navigation system manufacturers, ADAS manufacturers and map database suppliers (http://adasis.org).

The current public version of this standard is v2. In this version, the ADASIS Horizon is made available as a list of distinct paths that can be produced moving ahead of the vehicle. There is a main path which starts at the vehicle position, and secondary paths that depart at some point of the main path (Figure 4). Tertiary paths may depart at some point from the secondary paths (and so on).

These paths are produced by a software interface module (the Horizon Provider) that pre-processes the map database to show the vehicle the paths that it can follow from the current position up to a given level of bifurcations (many applications, including the AdaptIVe co-driver, uses only the main path).

Pre-processing the road network in a way to show paths ahead of the vehicle, simplifies the design of Driver Assistance applications, as they do not then need to discover the possible paths in the road network by themselves.

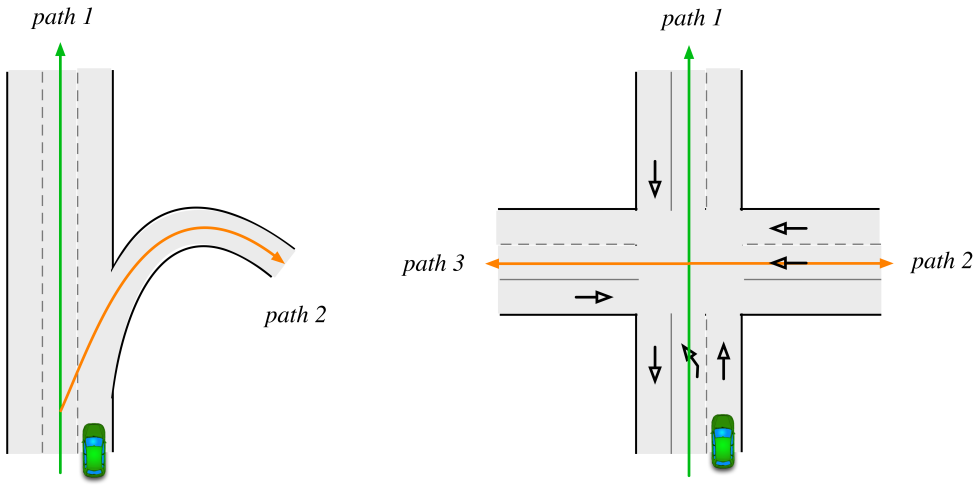


Figure 4: ADASIS v2 description of paths available for the vehicle. Path 1 is the main path. Paths 2 and 3 are paths departing from the main path at some point.

Every path is a simple data structure that describes various features of the road by means of interpolating functions: for example, the latitude and longitude of the path line is sampled at various longitudinal abscissas. Other points along the path can be determined, given the interpolating function. The collection of the sampled values, together with the interpolating function are called “profiles”. In addition to longitude and latitude, there are profiles for altitude, speed limit, heading, curvature, slope, number of lanes in both directions, and a few others.

The profiles of different features may be sampled at different abscissa. For example, the speed limit is piecewise constant (interpolation order 0) and sampled only at points where it changes. The path curvature is typically piecewise linear interpolated (hence describing the road with a sequence of clothoids).

The coordinates of the centreline, the heading angle and the curvature are (almost) redundant information. They are linked by the following equations:

$$\frac{dx}{ds} = cos(\theta), \frac{dy}{ds} = sin(\theta), \frac{d\theta}{ds} = \kappa \quad (1)$$

Where s is the curvilinear abscissa, $x = x(s), y = y(s)$ the coordinates of path point at s, $\theta = \theta(s)$ the heading angle and $\kappa = \kappa(s)$ the curvature.

Hence, in principle, one path could be reconstructed by integrating the curvature twice, with given initial heading and point $x(0), y(0), \theta(0)$. However, due to errors in the map database (and coarse data discretization) the two are rarely consistent (i.e., there can be errors between the location of the path derived from curvature integration and from interpolation of the latitude/longitude pairs).

As shown in Figure 4, ADASIS v2 describes *road* segments. Several authors have noted that for automated driving the description is not sufficient [9], [10]. For example, in ADASIS v2 there is information about the number of lanes in both directions but there is no information about the lane width of each lane, side banking, or information to precisely reconstruct the geometry of the lanes where they merge or divide.

A "lanelet" approach has been proposed in [10]. The road network is described with a graph of connected "lanelets". Each lanelet is modelled with its right and left edges given by polylines. Adjacent lanelets share the same polyline edge. Additional features are stored for each lanelet, which define traffic regulatory elements, including the position of the stop lines, which is important for intersections (and not present in ADASIS v2). A graphical editor and a C++ library are available for using lanelets models. However, this model is similar to a map database and needs to be queried to obtain the paths (here called corridors) that one vehicle might follow. In addition, to obtain the centreline of the corridor as a smooth function (which is helpful for trajectory planning) additional computations are necessary and, above all, there may be situations where the lane margins are asymmetrically distributed with respect to the smoothest central line; e.g., where lanes merge, one of the margins may vary suddenly and the lane in the middle of the two sides is not the smoothest possible trajectory (the trajectory a human driver would follow).

For Dreams4Cars, an extension of the ADASIS v2 is planned. Albeit lanelets might be used to model a road network, the description of "paths" ahead of the host vehicle is preferable at the tactical level, in particular because such paths simplify the discovery of trajectory affordances. Since ADASIS v3 is not public yet, Dreams4Cars will extend the ADASIS v2 model by including all the features that are necessary for producing an unambiguous detailed geometric description of lanes and roads.

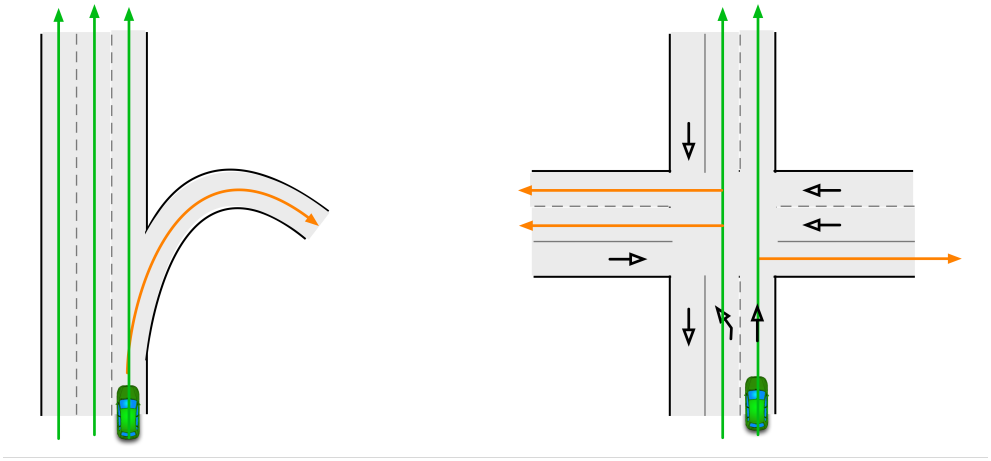


Figure 5: Extension of the notion of paths to the lane-level, which is planned for Dreams4Cars.

In particular, the following extensions are planned:

a) Dreams4Cars will adopt a lane-level approach: every lane will be associated with one path (Figure 5). Paths will merge or divide where lanes merge or split. Adjacent lanes will be modelled with adjacent paths.
b) Each path is determined by its initial point and heading $x(0), y(0), \theta(0)$ and by a profile of curvature $\kappa(s)$; hence path centreline and heading are determined by integration of (1); Figure 6.
c) For every lane/path, a curvilinear coordinate system is defined formed by longitudinal (s) and lateral position (n). Following [11] the conversion between curvilinear coordinates (s, n) and Cartesian coordinates (x, y) is obtained as follows (Figure 6). Using a curvilinear coordinate system simplifies the semantic interpretation of the movement of the vehicles.

$$\begin{aligned} x(s,n) &= xc(s) - n\sin\big(\theta(s)\big) \\ y(s,n) &= yc(s) + n\cos\big(\theta(s)\big) \\ \psi(s) &= \alpha(s) + \theta(s) \end{aligned} \qquad (2)$$

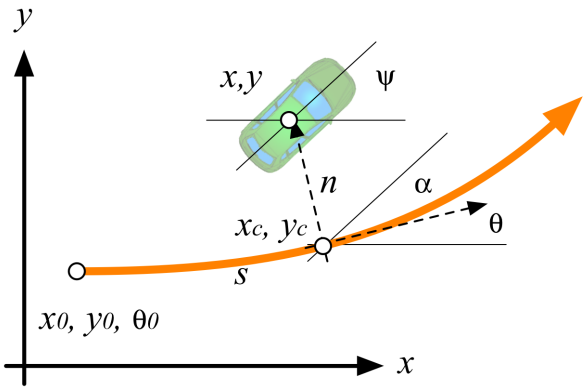


Figure 6: Coordinate systems: Curvilinear coordinates (s, n) and Cartesian coordinates (x, y). See equations (2) for conversions. x_0, y_0, θ_0 are the initial point and heading of the path, $xc(s), yc(s), \theta(s)$ the coordinates and heading of path point at abscissa s, as resulting from integration of (1), α is the relative heading and $\psi$is the absolute heading.

d) For every path, two profiles for the lateral distance of left and right lane margins will be defined (the lane margins may not be symmetric in merge zones).
e) For every lane, a profile that points to the adjacent lane is provided, if there is a lane; if not, the distance to the road border or curb is given.
f) Altitude, slope and banking profiles are given.
g) Regulatory profiles are given: speed limit, position of stop lines, regions to be maintained clear (where vehicle cannot stop, such as over pedestrian crossings, railroad crossings, etc.), position of pedestrian crossings, yield or way rights, traffic signs, directions of lane traffic, types of lane markings, position of traffic light etc.
h) Finally, the paths originating from the position of other vehicles will also be given (such as observing the road from the other vehicle's point of view). This is necessary for prediction of other vehicles' intentions (Figure 7). These paths may also be received with V2V communications (in which case the other vehicle may declare its indented lane).
i) Special types of "obstacles" may also be introduced. These include: regions where the vehicle cannot drive, areas where the vehicle cannot stand still (the centre of an intersection till the stop lines or a pedestrian crossing could be modelled this way).

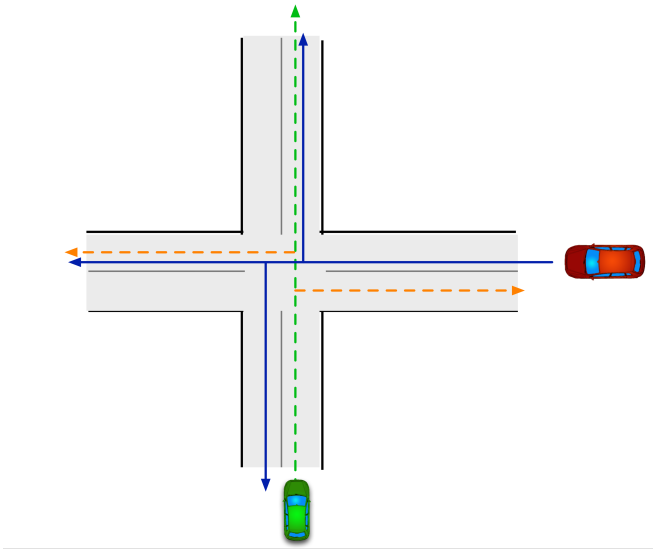


Figure 7: Knowledge of the paths available to other road vehicles is required for correct inference of their intentions and prediction of their paths.

3.3.5 Semantic annotation

The system will log high-level intentions/actions at each time interval using an appropriate hierarchical breakdown reflecting the nature of the perception-action (PA) architecture. The system will also generate and log labels during operation to describe the traffic context in which the system is currently operating (i.e., the broad road configuration described in perception-action terms). A time interval is here defined as a temporal duration during which the discrete, high level road configuration remains unchanged (i.e. with no changes in car order, lane occupancy etc). It is thus typically of the order of seconds.

In general, intentional annotation amounts to identification of the most likely set of legal groundings of hierarchical PA sequences satisfying the imposed logical consistency constraints (derived from the Highway Code). Thus, the architecture imposes a symbolic/sub-symbolic divide; changes in metric relations that do not constitute a discrete configuration change (e.g. within-lane changes in car proximity) are not logged as annotation. The 'logical resolution' of PA predication/annotation will increase with proximity to the interaction space of the agent.

The system architecture will implicitly ensure that event-logging protocols are consistent with the later generation of exploratory learning instantiations of the PA architecture (section 6.4: Exploratory Learning)

The logic reasoning component responsible for high-level semantic annotation will initially deploy declarative fuzzy reasoning; however, research is being undertaken to implement this reasoning in a neural architecture (which would sit at the apex of the general Dreams4Cars neural architecture, ‘dorsal stream’ Figure 1). The in-situ annotation system will thus either derive from top-level predicates generated via declarative reasoning, or alternatively, from the output of selected neurons in an integrated neural system. Configuration annotation will be augmented with digital-map annotations.

3.3.6 Self-monitoring system

The self-monitoring system detects anomalies and failures in the vehicle dynamics and related sensors (WP2.3). For this, the system compares sensor data with predictions of internal models. Because of model and sensor noise, the two will never match exactly. Hence, to detect anomalies, the question of how much a prediction is allowed to diverge from what is measured has to be answered.

However, the threshold to classify a mismatch as an anomaly is not constant, but varies depending on the current performed action and the state of the system[3].

To deal with this, Dreams4Cars will extend the internal models with the ability to generate predictions of the expected error between models and sensors. The expected error (which will be context and situation related) will thus provide a scale for defining the mismatch threshold in probabilistic terms.

Modelling the error between models and sensors could be realized either by Artificial Neural Networks (ANN), like in previous work [12], or with Deep Neural Network (DNN) architectures.

When an anomaly is detected, the system may try to determine the kind of alteration or context change or failure that happened. For example, if the Dreams4Cars agent has forward emulators for both dry and icy roads, and an anomaly has been detected when using the dry road emulator, the system might try to determine whether the other emulator is appropriate. Hence, by testing all the forward emulators available the system might discover the kind of changes that occurred in the vehicle dynamics, and such a discovery may allow on-the-fly adaptation of the inverse models used for control in the motor system (WP2.3). As an alternative, the system might develop a context classifier on top of the forward model library, to be used for selection of the forward model most suited to the current context (for example a slippery road model might be selected by default when it is raining).

To sum up, two possible approaches could be implemented (mixed strategies are also possible):

a) One classifier is used to classify the current context. While this requires the classifier to be trained on data of different environmental conditions, it directly operates on data of environmental sensors, such as temperature and rain sensors (and maybe cameras if using a DNN).
b) Different instances of forward models could be trained for each of the environment conditions and are run in parallel while the agent is driving. In case of a mismatch of the currently selected model the system could compare the predictions of the other models and might select the one that has the lowest error and select the corresponding context.

Training of the whole self-monitoring system and its models will be done in the training phase in the cloud simulation after the forward models have been adapted.

[3] Both sensor noise and model accuracy may be situation dependent (e.g., a forward model might be less accurate near a gear shift for a large acceleration/deceleration).

4 Cloud environment features and parametrization of scenarios

This chapter deals with the cloud simulation environment, implemented in the OpenDS driving simulator. This is the "dream" state, in which a copy of the agent drives in a virtual world in situations that recombine (efficiently) the salient cases annotated in the vehicle environment. Optimization of the agent sensorimotor system and discovery of new actions takes place here.

The proposed open-source driving simulation environment, OpenDS, has continuously been developed by DFKI and funded by the EU in various projects since 2012. The development has been initiated in order to provide an evaluation tool for automotive user interfaces, as full-fledged driving simulation software is high in price and low cost simulators often lack of extensibility. Thus, the primary goal of this cross-platform implementation was to provide a basic visual driving simulation toolkit to the research community at little cost. Binaries and source code of the latest release (v4.5) can be downloaded from the official website www.opends.eu free of charge. Up to the present day, the number of users has grown to more than 4,000 where most of them are affiliated to academia or industry.

The software has been implemented entirely in Java and is based on the jMonkeyEngine[4] (jME) which has become popular in Java game development. jME provides a high-performance scene graph based graphics API, an OpenGL supported renderer (Lightweight Java Game Library) and the Bullet physics library which is in use by top industry developers. Bullet is a multi-threaded physics engine which allows mesh-accurate collision shapes and enables the application of forces such as acceleration, friction, torque, gravity and centrifugal forces during simulation. The support of several common model formats allows loading almost any 3D environment. Further features of the rendering system are support of different lighting options (per-pixel lighting, multi-pass lighting, Phong lighting, tangent shading, and reflection), texturing (multi-texturing through shaders), and the capability to model special effects such as smoke, fire, rain, snow etc. Supported post processing and 2D filter effects are reflective water, shadow mapping, high dynamic range rendering, screen space ambient occlusion, light scattering, fog, and depth of field blur. Nifty GUI integration enables an easy-to-use toolkit for designing platform independent graphical user interfaces within the rendering frame, which is used for menus and message boxes during simulation. Further features of the game engine are basic multimedia (image, audio, video, etc.) and game controller (e.g. steering wheel) support.

Since the very first version of OpenDS, the driving simulation has been improved and extended iteratively for more than five years. The requirements of the funding projects influenced the road map and pushed certain developments such that the software is able to match up to products of the commercial market. In contrast to commercial simulators, OpenDS benefits from the open-source idea and gained from contributions of its users worldwide. Today, the software provides a comprehensive extensible toolkit including basic driving simulator functionality (e.g. traffic, traffic lights, weather effects, engine and transmission simulation, etc.) as well as specific functionality (e.g. multi-screen video output, Oculus Rift support, and interfaces for connecting external data consumers, traffic simulators, multi-driver environments, eye tracker hardware, professional steering wheels, authentic car environments (CAN bus), and motion platforms) and a selection of validated ready-to-use driving tasks [13].

The simulation environment is capable to load road network description files (XML format) which contain an exact description of the static scenario including roads, signs, traffic lights, buildings, and vegetation; of the dynamic scenario (e.g. vehicles, pedestrians, obstacles, weather conditions etc.); and a list of pre-defined events that might occur during simulation. By the help of these files, connected input devices (steering wheel, CAN bus controller, etc.) and output devices (screen output, motion seat, force feedback, etc.) can be configured, communication to external applications can be managed, and what data to be logged can be defined (c.f. Figure 8). Log data can be stored in text files and data base tables. Furthermore, the integrated Jasper Report module allows creating templates for visual representation (graphs, charts, etc.) of this data which can finally be exported to PDF format. In addition to video recording of the simulation, vehicle trajectories can be recorded for any later analysis of the driven pathway at any point of the simulation. At this point, the recorded tra-

[4] http://www.jmonkeyengine.org

jectory can be compared to a pre-defined normative trajectory in order to compute the deviation, which can be considered as a measure of driving performance.

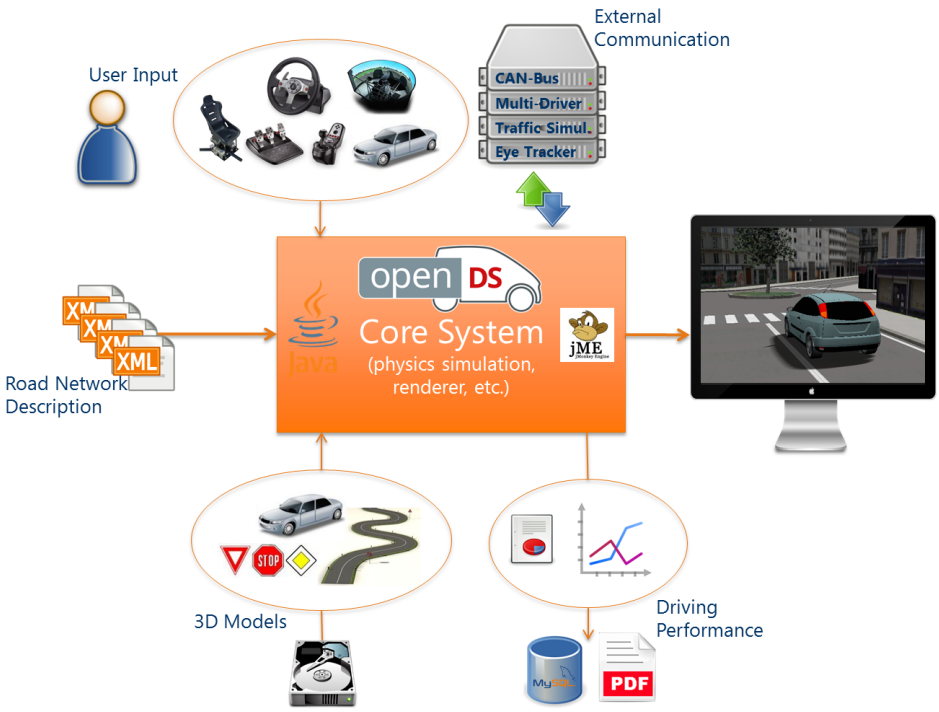


Figure 8: Road Network Description files used to set up the scenario, input devices, external communication, and logging.

For the scope of Dreams4Cars, there may be features mentioned before which will not be required; however, other features which are not available yet need to be implemented. Thus, we investigate in the following (in order of priority) in more detail the requirements and required extensions of the cloud simulation environment concerning vehicle dynamics model, environmental parameters, agent interfacing, generation of road networks, vehicles and pedestrian modelling, simulation of sensors, and simulation of vehicle-to-x communication.

4.1 Vehicle Dynamics Model

The current version of OpenDS uses the basic VehicleControl class of the jMonkeyEngine framework, which in turn uses the btRaycastVehicle model of the Bullet physics library. The vehicle is modelled with the dynamics of one single rigid body and allows setting the mass, inertia tensor and position of centre of mass of the chassis. Wheels and suspensions are not modelled in details as separate bodies. Instead, wheel forces are computed by means of four vertical rays. The ray's intersection point with the terrain is used to calculate the suspension length, and hence the suspension force which is applied to the chassis [14]. The suspension accounts for the spring and damping forces. There are two coefficients for damping: one for spring compression, and one for spring relaxation (in a real vehicle, the compression damping is set much lower than the relaxation damping, which means, when the vehicle hits a bump, it will not be transmitted to the chassis, resulting in a smooth ride). In addition to these suspension settings, the model allows setting a maximum suspension travel as well as the exact axle and wheel positions.

The friction model in Bullet is implemented as separate forces applied to each wheel where the ray contacts the ground. For each wheel a constraint equation is used. Expressing the friction as constraints allows the constraint solver of the physics engine to compute the friction force on each wheel. In more detail, this friction constraint consists of an axis to act along (lateral wheel slip), a target slip velocity (which is zero) and a maximum force according to Coulomb's friction law [14]:

$$\mathbf{F = \mu N}, \quad \text{where F: maximum friction force, } \mu\text{: friction coefficient, N: normal force}$$

This model for tire forces is very simple if compared to tire models in the literature, as it does not allow to accurately model tire sideslip phenomena (see improvements below). A further disadvantage of the ray cast model is that rays are infinitely thin and show incorrect behaviour on some kind of surfaces compared to a torus-shaped tyre model (however this limitation is of secondary importance compare to the first). While a ray cast wheel projects through a crack in the ground geometry, a convex cast wheel would roll over cracks correctly (c.f. Figure 9).

A further restriction of the current vehicle implementation is that aerodynamic forces are not implemented (which might play some role at high speed).

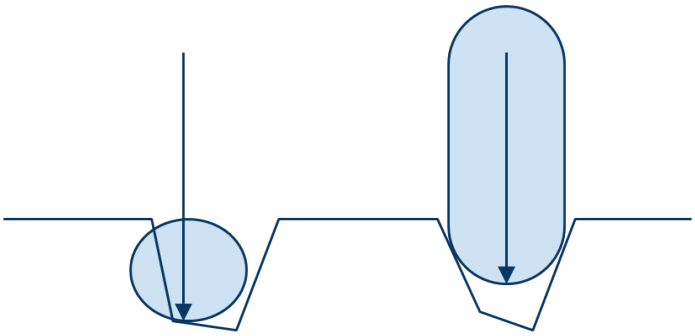


Figure 9: Ray cast wheel vs. convex cast wheel when hitting a crack in the road

Improvements. As the vehicle dynamics model plays an important role in the project, we will investigate to what extent the existing wheel/tire model can be improved within the integration work package. At least we need to ensure that understeering and oversteering are modelled properly, which means that suitable models for tire lateral and longitudinal forces should be implemented. Changes at this level result in adaptations of the physics engine, the game engine and the simulator source code.

4.2 Environmental Parameters

Environmental parameters denote all parameters that can influence the behaviour of the vehicle from the environment during the simulation. These parameters comprise: surface type (asphalt, sand, ice, snow, etc.), weather conditions (fog, rain, snow, etc.), and light conditions (sunrise, dark, bright, etc.). Within the scope of this project, there is no human-simulator interaction and, therefore, there is no need to simulate environmental parameters that only have impact on the visual experience (e.g. falling rain drops, sun light from the front, etc.). Instead, we focus on parameters influencing the physical behaviour or any other signal perceived by the artificial co-driver. For example, friction effects between wheels and road surface, or sensor effects that typically occur at certain light or weather conditions.

Improvements. In order to simulate different surface features, first the simulation needs the information of what specific 'material' is present at each wheel contact point. Since the current implementation does not provide comprehensive capabilities to deal with semantic information about the environment, this will constitute an important extension of the simulation software. In the current state, the driving scenario is generated by loading various 3D-models of environmental objects and arranging them in the scene. The simulator only has restricted information about scene objects and cannot differentiate between e.g. road and off-road surfaces. A basic solution to simulate surfaces with different adherence has been implemented in the form of a grip table which allows assigning grip coefficients to each scene object at mesh-level, i.e. for each mesh a specific grip value can be looked up in that table and be used for the wheel located at the given mesh. Figure 10 depicts the visual representation of such a table, where road, pavement, and off-road meshes have been assigned with a very high, high, and medium grip value, respectively.

If a local discontinuity of the grip is caused by weather effects (e.g. wet road section caused by rain), a synchronisation of the physics simulation with the visual appearance (e.g. raindrops, puddles, etc.) might be con-

sidered to increase realism for the visual representation of the scene. However, as the co-driver does not derive the information (at the moment) from the image of the front camera visual extensions have low priority.

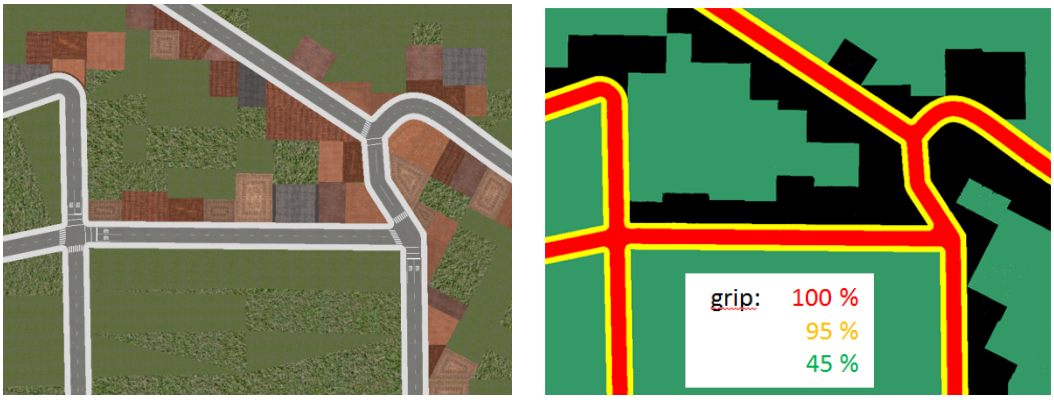


Figure 10: Manual annotation of friction slip at mesh-level

Environmental parameters that influence sensors. In addition to environmental parameters which influence the physical behaviour of a simulated vehicle, further parameters that could potentially disturb the sensors (radar, ultrasound, LIDAR, etc.) of a sensor-equipped vehicle need to be considered in the simulation as well. Typical environmental effects that can reduce the quality of the sensor data and need to be implemented are fog, rain/snow, and adverse light conditions. As mentioned above, this does not include the visual implementation (which is already available for fog, raining, and snowing). However, it must be recalled that Dreams4Cars is not intended to develop sensor technologies; hence the influence of environmental conditions on sensors has to be considered only as for what concern the perception effects (not necessarily all the details that might be need for optimizing sensor designs).

4.3 Interfaces

OpenDS provides an implementation of a TCP server which can connect with external clients. Once a client is connected, it can subscribe to be updated about any of the following car parameters: accelerator pedal state, brake pedal state, steering wheel angle, headlight state, selected gear, rpm, fuel consumption, latitude, longitude, altitude, heading, and speed. As the simulation has full access to every parameter of each scene object, the interface can simply be extended by any parameter allowing it to be consumed by the client. A comprehensive list of parameters (including vehicle data, sensor data, environmental data, traffic data, infrastructure data, control instructions (remote setup of scenario)) needs to be derived from the definitions of section 3.3.3. Alternatively, the server can also support UDP connections, if fast data transfer has higher priority than data loss.

The CarMaker environment (Figure 1) can be replaced to the OpenDS simulation environment for the Quality Assurance phase using the same interface.

4.4 Generation of Road Networks

The generation of road networks can be considered as a two-level task: in the first place, we need to generate a 3D-model consisting of polygons providing the ground to drive on; in the second place, we need to enrich that terrain information (low-level data) with semantic information (high-level data) representing lane configuration, traffic pathways, traffic lights, etc. Adopting a road description format as close as possible to emerging standards (such as ADASIS) is recommended for modelling the road network.

Regarding the first job, there are plenty of 3D modelling tools available for static scenery generation; however, in Dreams4Cars we need to explore methodologies to dynamically generate rule-based scenes (driven by various dreaming mechanisms).

While usual driving simulation use-cases are built on a fixed set of re-usable road networks, simulation in Dreams4Cars ideally consumes a large amount of slightly different scenarios generated on-the-fly.

The way to generate scenarios for OpenDS may be based on a) manual modelling with a 3D modelling editor or b) based on importing real-world data from OpenStreetMap[5] (e.g. OSM2World[6]) or c) based on road network generation with a specific graphical editor (e.g. Esri CityEngine[7]). Manual modelling is the most flexible approach; however, since it requires lots of human resources and cannot be automatized, it will not be a feasible solution in this project. Importing scenarios from real-world data is a rather convenient way to collect large amounts of road configurations: it may be a viable solution but it is not a genuine dreaming mechanism because it will not generate any conceivable network. The third approach, generic scenario generation, is the most promising one, as it uses tools that are able to turn road network descriptions into 3D models. Usually, road network modelling editors comprise a graphical user interface and require some manual input. For the scope of Dreams4Cars, however, the design of the road network will be proposed automatically in order to create salient situations from previous situations experienced in real driving.

Editors with good prospects that have been considered are Esri CityEngine and the new Scenario Editor of IPG CarMaker 6. The latter one is rather attractive, because CarMaker will be used for quality assurance. However, a close look at the scenario description format used by CarMaker, which is named ROAD5, revealed that the format lacks of a documented specification. Furthermore, the XML files of this format contain paths to internal CarMaker resources that are only available as binary files. In addition, to our knowledge there is no standardization activity promoting ROAD5.

Conversely, CityEngine supports several common 3D model formats and allows for rule-based road network generation. The computer is given a code-based procedure which represents a series of geometric modelling commands which then will be executed. Instead of the classical intervention of the user, who manually interacts with the system, the task is described abstractly, in a rule file [15]. However, it still remains to be investigated whether CityEngine can be driven from a road network description of the co-driver agent or vice versa.

Alternatively, the non-commercial toolkit OSM2World might be worth to be considered. This tool is capable to convert road networks described in OSM format into 3D models. This format is based on XML and contains basically nodes (with latitude and longitude) and ways that connect several nodes. Furthermore, the format allows annotating various tags to single nodes and ways (e.g. road width, number of lanes, etc.).

To sum up, if we succeed to describe the road layout to be created as a list of geographic positions (latitude, longitude, altitude, which may in principle be derived for the ADASIS emerging standard), an OSM file could be generated which in turn could be converted to a driving scenario. Issues of this solution may be incorrect representation of lane markings, inaccurate transitions from n to n+1 lanes, and imprecise surface features.

OpenDS provides several tools to add additional objects to the scenario, to define traffic pathways, and to assign a mapping of authentic GPS-coordinates. Usually, after a 3D model has been created, the Object Locator is used to add road signs, trees, traffic lights, etc. to the scenario. These objects will not be hardwired to the 3D model, but rather the model and objects will be referenced in a road network description file which can be executed by the simulator. Using several road network description files allows re-using the same model with different road sign configurations.

Improvements. Currently, the road network description format in use is a custom implementation considering the minimum set of supported features. In order to provide more detailed semantic information to the simulation environment, the interface needs to be extended or replaced by an open standard (e.g. ADASIS, OpenDRIVE, RoadXML, etc.). Furthermore, the Object Locator, which uses a graphical user interface to place objects in the scene, needs to be adjusted in order to support automatized scenario generation, i.e. objects need to be placed without human interaction according to the information collected by the agent in the wake state. Analogously, dynamic objects and the way they are intended to move by the dreaming mechanism need to be included to the road network description in forms of initial configuration, trajectory, and speed profile. Further-

[5] http://www.openstreetmap.org

[6] http://osm2world.org

[7] http://www.esri.com/software/cityengine

more, semantic information like lane configuration (intersections, neighbours, n-to-n+1-transitions), traffic light and speed limit affected lanes, and relative vehicle positions need to be implemented.

4.5 Vehicle and Pedestrian Modelling

As a specific kind of semantic information, traffic could be added to a scenario, i.e. computer-controlled vehicles and pedestrians. Describing trajectories of traffic objects relative to the simulation time frame is a highly relevant simulation feature required in Dreams4Cars in order to exactly recreate situations experienced in the wake state and variations thereof.

For this purpose, OpenDS provides basic annotation capabilities as contained in the road network description standard ADASIS: links and nodes. A plain 3D road model can be annotated with a collection of directional links (also called 'segments') and nodes ('way points') that define connectivity between links. Primarily, a node consists of a location in the 3D coordinate space, however, further semantic information like light states (turn signal, brake light, head light), close-by traffic lights, and whether a node requires a full stop or even waiting time (e.g. stop sign) can be attached to each node and will be applied to approaching vehicles and pedestrians (if applicable).

A traffic object can be assigned to one of the way points where it will start from at the beginning of the simulation. It can move from one way point to another if both are connected by a unidirectional segment originating from the one-way point and targeting the other way point. In order to allow a vehicle to return to the previous way point a separate segment pointing in the opposite direction would be required. As soon as a traffic object reaches the end of a segment (i.e. approaches to a node), the subsequent segment will be chosen randomly from all outgoing segments of the node according to their probability values. Outgoing segments of each node are annotated with individual probability values which sum up to 100%: the higher the individual percentage of a segment, the higher the probability that it will be followed by a vehicle. Furthermore, traffic objects can be equipped with individual lists of preferred segments, which will overwrite the default selection according to probabilities. By the help of this concept, individual trajectories of traffic objects can be defined. Besides the probability value, each segment can have further properties like: maximum speed a traffic object is allowed to drive/walk at when following this segment, a pair of neighbour segments (left/right) used for overtaking procedures on multi-lane roads, a list of segments having higher priority than this in case of intersections, and jump markers. A jump marker indicates whether a segment is an ordinary segment that can be followed by any vehicle/pedestrian or if this segment is used to connect two distant nodes making the vehicle disappear at the originating node and instantly appearing again at the target node. Jump segments can be used to avoid deadlocks, e.g. prevent randomly moving vehicles from getting caught in a dead-end road.

In order to enable proper vehicle and pedestrian simulation, at least two independent networks must be created – one for vehicles and one for pedestrians. Connecting a node from the vehicle network with a node from the pedestrian network would result in cars driving on the pavement and pedestrians walking on the road, as traffic objects can freely move along connected segments. Furthermore, vehicles automatically stop in front of red traffic lights and keep a minimum safety distance to all other traffic.

Improvements. Figure 11 depicts a graphical network generator application which has been developed for fast and efficient manual network creation based on a top-down view.

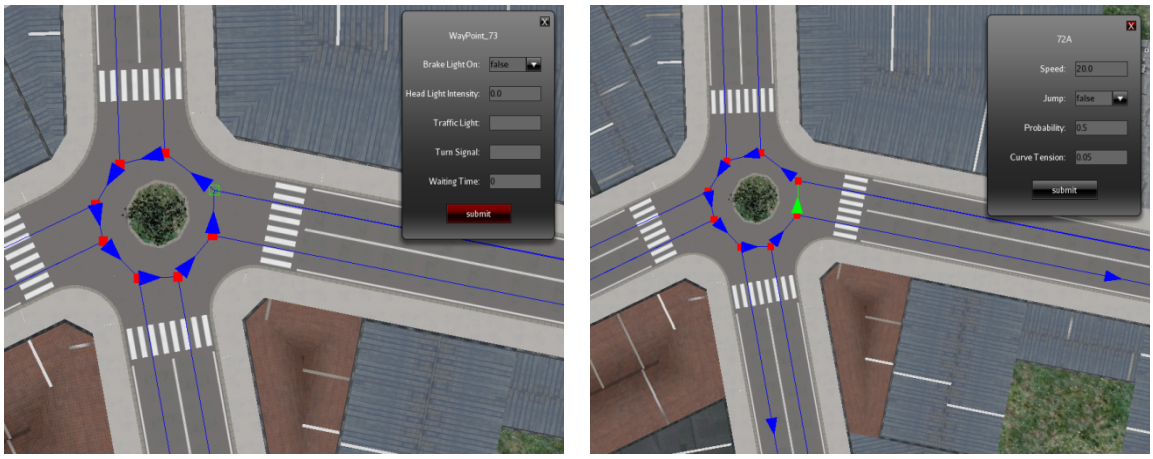


Figure 11: Network generator: way point editing (left) and segment editing (right)

This graphical editor allows loading empty scenes as well as scenes containing complex road networks consisting of many way points and segments. The user can add, change, and remove way points and sections as well as set up traffic objects (type of car/pedestrian, mass, maximum speed, acceleration/brake force, traffic observation distance, etc.), define their initial positions, assign neighbour and priority segments, and store all changes to a format that could be loaded by the simulator. Based on the functionality of this editor, an algorithm to place way points and connect them by segments in a reasonable way according to the road layout designed by the dreaming mechanism is required. For the scope of Dreams4Cars, the road network description (c.f. 4.4) must be extended by traffic information in order to create appropriate scenarios, on the one hand, and appropriate traffic networks, on the other hand. A feasible solution could be adopting ADASIS, OpenDRIVE or RoadXML.

4.6 Simulation of Sensors

The perception system of the co-driver may use short range and long range radar, cameras, LIDAR, and ultrasound sensors (which exist in the test vehicles). In the simulation environment, the same sensors must be simulated. Data for all scene objects like velocities, accelerations heading angles and any other state variables are available during runtime of the simulation. Hence the high-level output of sensors (such as directions and distances and bounding boxes of objects) can be easily computed replacing the high-level output of the real sensors when the co-driver operates in the simulator. Conversely, raw sensor data (for example LIDAR scans) requires models of the sensors and may not be available or may be available in simplified ways[8] (e.g., as ray traces for the LIDAR, see below).

On the other hand, providing camera images is a basic feature of each visual driving simulator and can naturally be provided from every position –either moving or static– with almost any resolution. Hence, simulation of cameras is available as raw pictures. However, if a cloud service were to be set up (which is a possible follow up product of the project), providing camera images requires the presence of a GPU, which is not part of the basic package offered by most cloud-providers, and is limited by the capabilities of the network connection in case the image data has to be streamed to a remote server.

Concerning the simulation of a Global Positioning System (GPS), OpenDS provides capabilities to convert model coordinates to real-world coordinates and vice versa, allowing the co-driver to request authentic GPS sensor data (latitude, longitude, altitude, heading) from beforehand geo-referenced scenarios.

Improvements. Concerning the simulation of sensors at raw data level, in order to simulate radar, LIDAR, and ultrasound sensors, different ray emitters have to be attached to the virtual vehicle where they cast rays in the respective directions to scan the environment. Scanning rate, resolution, and range need to be adjusted to the

[8] Dreams4Cars is not intended, in the current project, to be a testbed for sensors (albeit it might serve this function if accurate models of sensors are integrated).

typical capacity of the respective sensor and realistic failure and distortion of the sensor data (e.g. fog, raining/snowing, and adverse light conditions) need to be modelled adequately.

Figure 12: Mockup top-down view (left) of the current scene (right).

Furthermore, rendering techniques of the simulator could be used to capture a top-down view of the scene at the current position of the vehicle in order to create additional image-based sensor data or to monitor traffic in the vicinity. Figure 12 depicts how the monochrome top-down view of a typical traffic scene could look like.

4.7 Simulation of V2X

In addition to sensor data, vehicle-to-x (V2X) communication will provide an important source of information to the co-driver agent. For the scope of Dreams4Cars, test sites and vehicles will be equipped with V2X hardware allowing vehicular communication which is restricted to infrastructure (V2I) and other vehicles (V2V). Communication involving pedestrians or other devices is out of the scope of the project. Due to interchangeability of the vehicle and simulation environment, the cloud environment needs to mimic the communication of the vehicle environment.

As the driving simulator has full access to all scenario objects during the simulation, opponent vehicle parameters (position, speed, planned route, brake status, etc.) as well as infrastructure parameters (e.g. in case of a traffic light: position, affected lanes, light state, remaining time till next light change, etc.) can be provided to the co-driver agent.

Improvements. Suitable protocols for communication must be agreed on and implemented at simulator end. In the context of the AdaptIVe project, various protocols have been evaluated such as CAM (vehicle status broadcast), MAP (topological definition of lanes), and SPaT (signal phase and timing of traffic lights). Available semantic information needs to be translated to the message format of these protocols. Furthermore, filters have to be applied in order to limit data reception to vehicles/infrastructure in the vicinity of the co-driver. A typical ad-hoc network for V2X communication can be simulated by “disconnecting” senders that exceed the characteristic operating range of wireless LAN.

5 Vehicle environments and test sites

5.1 Test sites

5.1.1 Test sites in Germany:

DFKI contacted different companies and intuitions in order to organize the most appropriate test track. The selected test sites are following:

- Traffic Training Centre Bremen (Verkehrübungsplatz Bremen)[9]
- ADAC Training Centre Bremen[10]
- Aldenhoven test centre[11]

5.1.2 Traffic training centre Bremen

Figure 13: Test sites in Bremen
(left: traffic training centre Bremen, right: ADAC training centre)

The traffic training centre in Bremen (see in Figure 13 – blue bordered area) serves as test environment for people, who basically want to improve their driving skill in a realistic environment. For this purpose, the train-

[9] http://www.verkehrsuebungsplatz-bremen.de/

[10] http://adac-weser-ems.de/fahrsicherheits-training/

[11] http://www.atc-aldenhoven.de/en/

ing centre is a 1.200 m long road built with various surfaces (e.g. asphalt and cobbles surface, tram rails, hill), including 31 parking lots as well as basic traffic and environmental objects (e.g. road sings, traffic lamps, trees etc.). There are also different cross roads (intersections – roundabout) to practice different realistic drive situations (e.g. right of way or other traffic rules) with other road users. The road lanes and other road markers enable different training scenarios for the agent with up to 30 km/h speed limit (see in Figure 14). This test site is to be rented for up to 5 hours per day according to the test plan.

Figure 14: Pictures from traffic training centre in Bremen

5.1.3 ADAC Training Centre Bremen

Another test facility located in Bremen is the ADAC training centre (see in Figure 13 – yellow bordered area), which is used for the purposes of gaining expert-level driving skills, for non-professional and professional drivers. The test centre has different test and training areas, which will be used to have reproducible controlled anomaly case for the training and evolution of the agent. A steel sliding surface (20 x 60 m) and circular path (Ø 53 m) serve different realistic scenarios to realise alteration/context change/failure of vehicle components as well as other anomalies. This test site will be rented per day according to the test plan.

5.1.4 Aldenhoven test centre

The Aldenhoven test centre (ATC) (see in Figure 15) is an interdisciplinary test field, used by car manufacturers and suppliers for development and validation of vehicles, vehicle components or vehicle functions. The eight track elements (oval circuit, rough road, vehicle dynamics area, intersection, braking test track, handling track, hill section and highway) provide different scenarios for tests. The intersection part of this test area is under construction for testing the autonomous driving skill of a vehicle in a controlled, realistic, urban environment. On this intersection, smart infrastructures and V2X communication with environmental objects will be available. It is planned to use this track for reproducible driving scenarios when it will be operative (late 2018).

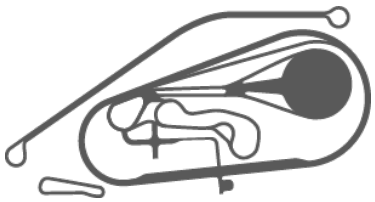


Figure 15: Aldenhoven Testing Centre track elements

5.1.5 Test site in Italy

CRF will perform tests for autonomous driving in the test track of Safety Centre located in Orbassano, close to the CRF location. The test track consists of a two lanes irregular-ring road of about 1700 m length. Some of the scenarios addressed in the project will be reproduced in this test site, mainly to verify the level of coherence between the simulation and the demonstrator vehicle test environments.

5.2 Test vehicles

5.2.1 Mia electric car:

The Mia electric car (see in Figure 16) is an off-the-shelf electric vehicle, which is modified with x-by-wire control to permit autonomous driving. The technical specifications of this vehicle are summarized in

The customization involves four subsystems, which are throttle, brake, steering and driving direction (i.e. forward and reverse). These are accessible for autonomous driving. A Kontron mobile embedded PC with an Intel Core i7 core is integrated on this vehicle as a high-level control unit. The interfaces of the vehicle, after customization, are listed in Table 4.

For perception, different types of sensors are integrated, as shown in Table 5.

There are three different physical communication bus types on this vehicle, which provide the communication between low-level vehicle control and high-level control (CAN), sensors (Ethernet, USB) and high-level control unit visualised in Figure 17.

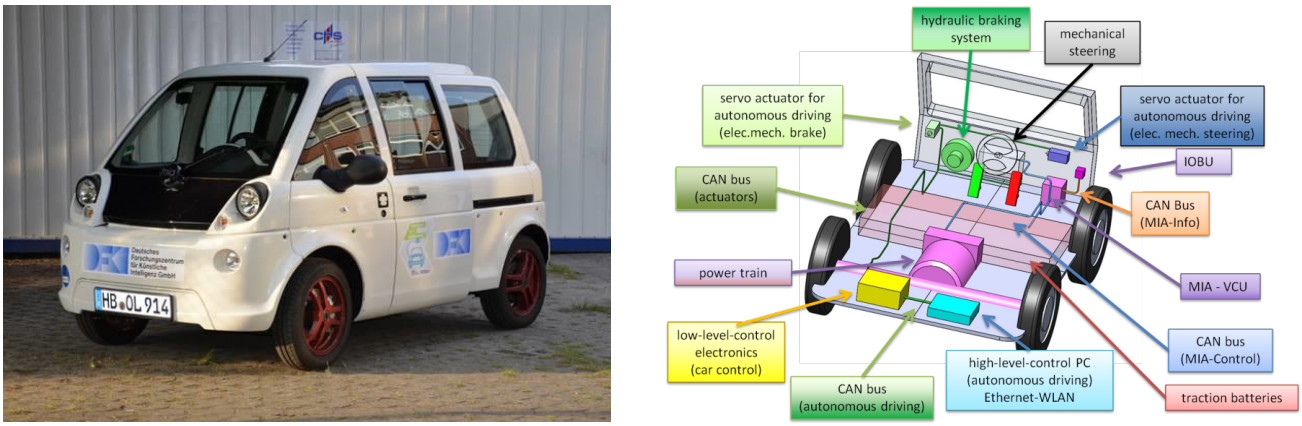


Figure 16: Mia electric car and the principal illustration of MIA components.

year of manufacture:	2012
class:	minivan
engine:	electric motor 24 kW
max. speed:	110 km
acceleration (0–50 km/h):	8,3 s
turning radius:	8,5m
electric engine battery:	8 kwh – LiFePO4
range:	~ 60-80 km
charging time:	ca. 3 - 5 hours (230V-AC, 16A)

Table 3: MIA technical specifications

Vehicle information	Electric mechanical interfaces	low-level & high-level interfaces
MIA Car information CAN bus channel with IOBU	steering	CAN bus
	brake	Ethernet (front / rear)
	throttle	USB interface (front / rear)

Table 4: MIA available interfaces after customization.

Sensor type	IMU + GPS	LIDAR	Laser scanner
Sensor name	Xsens MTi-G-700	Velodyne HDL-32	2xHokuyo UTM-30LX-EW

Table 5: The external sensor setup of Mia electric vehicle.

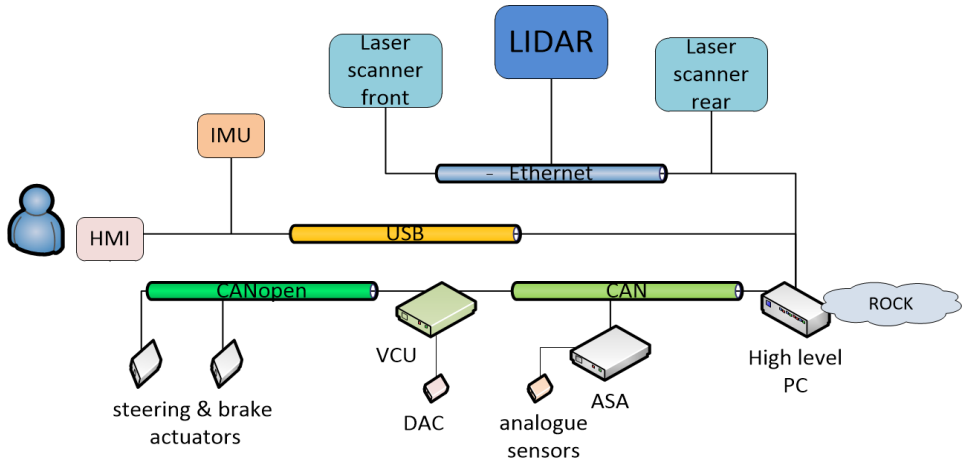


Figure 17: In-vehicle networks for control and sensor communication.

Currently, there are C++ drivers in ROCK framework[12] for sending commands to the car and receiving its status back. The commands include:

- Throttle
- Steering angle
- Brake
- Drive mode (forward, reverse or neutral)
- Control mode (manual, autonomous and neutral)

The status information of vehicle includes:

- Throttle position
- Steering angle
- Brake position
- Drive mode (forward, reverse or neutral)
- Control mode (manual, autonomous and neutral)
- Emergency button state
- Total distance travelled
- Wheel absolute positions (left and right)
- Raw average wheel speed
- Filtered average wheel speed of front wheels
- Vehicle speed from filtered average wheel speed
- Vehicle acceleration from vehicle speed

The driver task communicates with the hardware via CAN bus (see in Figure 17). Furthermore, the task converts the desired commands coming as input in the ROCK ports to CAN messages and sends it to the Vehicle Control Unit (VCU) (see in Figure 18). The latest status is send back by the VCU, which is read by the driver and written on ROCK output port as status. In future, this driver will be ported to ROS framework.

Apart from the driver, there is also a basic controller for receiving the desired speeds (forward and turn) and tries to maintain the commanded speed using the throttle, brake and steering commands, which are in turn sent to the driver. It has a design of a PID controller, where the positive output values serve as throttle command and the negative values serve as brake. The throttle and brake commands are also scaled differently. The controller works satisfactory for car velocities below 30km/h. The car model and the controller require

[12] www.rock-robotics.org

further development and testing at higher speeds. Furthermore, in order to control the car more effectively on slopes, the data from IMU is introduced to the system.

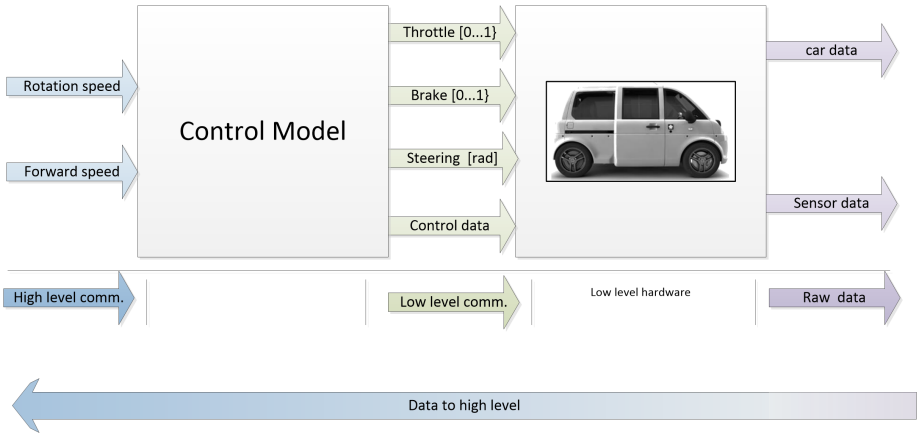


Figure 18: Mia control model and communication between control layers

5.2.2 Jeep Renegade

CRF will use in Dreams4Cars the demonstrator vehicle based on Jeep Renegade developed in the AdaptIVe European project. The demonstrator vehicle will be updated in the equipment in order to be used inside Dreams4Cars.

Figure 19: CRF AdaptIVe demonstrator vehicle on Jeep Renegade

Currently, the vehicle is equipped with different sensors and information sources to detect the environment: front radar, lidar and camera, side ultrasound sensors, blind spot radar sensors, GNSS receiver and Electronic Horizon.

Information coming from these sources is combined and organised in the fast prototyping unit (dSPACE MicroAutoBox) in order to generate the scenario description sent to the co-driver module running on a CarPC (Sintrones VBOX).

The output of the co-driver describing the manoeuvre to be followed is then sent back to the fast prototyping unit where it is used to control the vehicle through the vehicle actuators (steering, brake, engine, automatic gearbox). Relevant information about what the system is doing is also given to the driver through a dedicated HMI display; moreover, the driver can interact with the system using activation buttons on the steering wheel.

6 Generation of dreams

As already mentioned, the cloud simulation environment is implemented as an OpenDS simulation, i.e. the dream state. While the dreams are not an exact equivalent of the dreams of biological agents, there are some crucial aspects of "biological dreams" that will be drawn upon in this project: (1) dreams and other kinds of mental simulations are off-line, i.e., they are active but are not interacting with the controlled entity, the extra-neural body in biological agents, in particular (2) dreams reactivate the control system as if it were interacting with the controlled entity. (3) Dreams enable the biological agent to think about previous and future situations to (4) increase its ability to handle situations in the waking state [16], [17].

6.1 Creation of imaginary scenarios

The dream-state requires a number of functions, which provide some constraints on how the co-driver will operate and on the overall architecture. The functions of the dream-state amount to the following types of simulations:

- Simulations of previous events.
- Simulations of entirely novel events.
- Simulations that recombine aspects of previously encountered events into new events.
- Goal directed simulations that explore different simulation paths to the same goal.
- Goal simulations that opportunistically explore which goals can be achieved in a given situation.

6.1.1 Previous event simulation

Simulating previously encountered events, i.e., replaying the experiences that match those of the waking, driving state, serves mainly as a validity and reliability check of the dreams. The successful reproduction of previous events thus function as an implicit validation of the overall mechanisms used.

The simulations of previously encountered events will be achieved by using the motor cortex output (after action selection) of the co-driver as the input to the OpenDS simulation environment. The simulation process may in this task be started by randomly picking a perceptual state from the run-time collected data and let the co-driver generate the motor cortex output. This behaviour output is used as input to the OpenDS simulator, which acts as a forward emulator that generates the perceptual input (e.g. in the form of multi-beam LIDAR scans, digital maps or any of the sensory inputs described in section 2.1) corresponding to the likely consequence of performing that action, which in turn becomes an input to the co-driver. In this way, a closed loop of forward emulation in OpenDS and the inverse models of the co-driver is able to generate simulated chains of behaviour of various length. In practice the co-driver interacts with the emulator instead of the real world, which should reproduce the real situation.

This type of simulation allows us to assess if the OpenDS simulator is able to function as a forward emulator with the co-driver as the controller and that the environmental simulation is sufficiently similar to the real driving environments. This includes testing if there is some limit to how long these simulations can be as previous work on robot models has experienced problems with increase of noise in the simulations.

This type of simulation requires that the OpenDS environment is somewhat similar to the real driving environments so that comparisons can be made with regard to generated behaviour from the co-driver and predicted perceptions. In this case, the forward model in the co-driver architecture (section 2.3) is not used for comparison but instead the OpenDS generated perceptual input and the collected perceptions from real driving experiences are compared.

While this first type of simulation is only for validation purposes, the following simulations serve the purpose of learning new behavioural strategies.

6.1.2 Novel event simulation

Simulations of entirely novel events will be established by reserving some of the real-world driving data for use in the OpenDS simulation only. The reserved data sets containing novel scenarios will be labelled to indicate their importance, for example because they represent situations that are accident prone, or near crash inci-

dents. These simulations, while generating novel behaviours, mainly serve as a useful first step for testing the learning of new behaviours in the dream-state. An advantage of this approach is that it necessarily leads to "reasonable" hypothetical situations since they do, in fact, stem from real situations, but it requires annotated data from real-driving situations. A main advantage could be had if the annotation could be made automatic and derived from several different sources of driving data.

Another approach to creating novel events randomly instantiates the simulation starting point to push the simulation towards interesting situations. The dreams can then start by having a different set of objects (e.g. type of vehicle, trucks instead of cars), the structure of the road (straight becomes curved), the traffic density may change, and other road users might show a different type of traffic behaviour. The situations that are created should not be completely random as it would take a very long time to simulate all possible transformations but the architecture should be biased towards nightmares, that is, the dreams should be about situations that would be a possible threat and cause of an accident. This is done by choosing situations that are known to be accident related or by choosing uncommon events. The uncommon events would be beneficial to learn in dreams as they are not easily learned in real driving (even though such situations could be created on test tracks). In practical terms these situations may be created by properly 'scripting' the OpenDS environment, by generating the parameters of section 4 (vehicles, road, environmental conditions, sensor noise, etc.) according to suitable distributions.

6.1.3 Recombination simulations

Novel events can also be achieved by rearranging percepts, that is, objects and other road users may change into something else. These re-combinations are achieved by random rearrangements of percepts. This includes for instance changing (1) the type of vehicle encountered in a given situation (e.g. an oncoming bicyclist on a narrow road is replaced with a truck on the same road), (2) modifying trajectories of road participants (e.g. a pedestrian previously seen crossing at a pedestrian crossing is now simulated crossing 100 meters in front of the crossing), (3) changing the type of environment (e.g. a high-way environment is replaced with a country road), (4) rotating between forward emulators (e.g., replacing the vehicle dynamic model for dry road with the one for wet road). Also in this case, it is desirable to bias these random re-combinations towards situations that are likely to generate novel and useful strategies in the same way as previously described. The application of global metrics of quality for the behaviour (section 6.2) allows us to discover these. Failing to correctly bias the simulations may induce learning of too cautious behaviours, such, e.g., the co-driver thinking that pedestrian will always cross outside pedestrian crossings in front of the car.

6.1.4 Goal directed simulations

Another type of simulation does not alter the perceptions per se, but instead explores the solution space for a given strategic task. In these simulations, a strategic goal such as 'turn left in the next intersection' is set and then the co-driver tries out different strategies to learn different strategies that lead to the goal, but also to discover and un-learn or decrease the salience of the actions that lead to accidents or undesirable driving such as jerkiness.

6.1.5 Goal exploration simulation

Subtly different from the previous types of simulations, these simulations generate random goals in a given situation and try to achieve them. Such simulations assist the discovery of what goals can and cannot be realistically achieved in given situations. For both goal directed simulations and goal exploration simulations it will be an exploratory task in itself to find the levels of the hierarchy of perceptual goals that are most advantageous for learning new behaviours.

6.2 Dream-generating mechanisms

To construct the simulations mentioned in 6.1.1-6.1.5 mechanisms of the co-driver itself is reused in the simulation state (i.e. inverse and forward models) and some mechanisms will reside in the OpenDS environment itself.

Co-driver output. The basic driving force of the dream generation in all of the simulations is the co-drivers motor cortex output of longitudinal and lateral control signals which is used as the main input to the OpenDS sim-

ulation, in a similar fashion as efference copies of motor commands (at various levels of the motor hierarchy) are thought to drive mental simulations in biological agents [19]–[21].

Environmental generation aspects. The first three types of simulation (6.1.1-6.1.3) can be achieved by changing aspects within the OpenDS car simulation environment, such as initialising the starting point to a particular set up (6.1.1 and 6.1.2) or replacing objects (6.1.3). As mentioned above and detailed below in section 6.2.1 how the simulations are changed can be achieved by resorting to the optimality criteria.

In the perceptual domain some advances has been made using convolutional networks to generate natural images from higher-level descriptions of the images such as style, viewpoint, and transformation parameters such as color, brightness, saturation et cetera [7]. Also, they have been able to interpolate different viewpoints to create previously unseen but actual viewpoints of the learnt objects. This type of mechanisms could possibly be an effective means to generate novel events and recombinations within the OpenDS environment.

Another method is to re-use the mechanisms of the co-driver as detailed in the next two paragraphs.

Behaviour generation aspects. The goal directed and goal exploration simulations is about exploring the motor space and we here explore different types of inverse problems approaches.

Top Down Exploratory Behavioural Instantiations of the subsumption architecture may be used to generate goal directed simulations by a Perception-Action learning system. Here, the randomized selection of perceptual goals within the hypothesized perception-action hierarchy. Thus, while the system is always "*motor babbling*" in a manner analogous to the learning process of infant humans, the fact that this motor babbling is carried out at the highest-level P-A manifold means that the learning system as a whole more rapidly converges on the correct model of the world. Higher-level percepts thus become the goal states of actions parameterised per hierarchical level. The creation of each perceptually parameterised action at progressively higher levels of the PA hierarchy arises from a generalization over the tested space of action possibilities followed by a parametric compression. This in turn permits active sampling of the perceptual domain – the agent can propose actions with perceptual outcomes the agent has not yet experienced, but which are consistent with its current representational model (this implicitly also guarantees falsifiability of the perceptual model).

Motor babbling via high level randomised instantiation of perceptual goal states thus implicitly specifies a task scheduling problem at the hierarchical level immediately below the level at which the task is specified i.e. the specification of an action selection problem. Thus, in the dream state, the predicatised format adopted enables progressive top-down randomised variable instantiation such that a series of increasingly-grounded hypothetical imperatives/goals are created. This, in effect, allows a number of different series of hypothetical situations (albeit in a 'non-representational' sense) that collectively provide training data significantly beyond that obtained from real-world sampling.

Motivated learning. Another mechanism that is used to explore different behavioural paths in goal directed and goal exploration simulations is so called motivated learning. Here, the agent takes advantage of opportunistic interactions with the environment to develop a sense of 'agency' - knowledge of what actions cause predictable effects in the environment, mediated by the construction of internal models of action-outcome pairing in the brain (see forward model below, and section 2.3). Motivated learning promotes changes in the action policy so that the agent repeats actions that produced novel or surprising environmental events. Thus, this mechanism can be reused in the dream state to control the dreams such that actions that generate novel environmental situations will be explored further until the action-outcome pairing has been learnt in the co-driver.

Co-driver based forward models. The forward models of the co-driver both guides what is learnt during dreams and is at the same time updated during dreams while new situations are explored and experienced in the dream-state. There are two ways in which the forward models of the co-driver can guide the dream experiences: (1) the prediction error of the forward models could be used to tag which situations should be tried out in the dream state (6.1.1-6.1.3), and (2) in the dream state, the forward model of the co-driver might play a role in motivated learning by using the difference between the forward model of the co-driver and the forward emulation of the OpenDS simulation.

6.2.1 Optimality criteria

The metrics for evaluating the behaviour of the system can also be used for developing mechanism to prime the dream state toward useful experiences to learn from. During real driving, data can be logged in relation to the following criteria: (1) safety measurements, such as space-time distances, (2) the degree to which traffic rules are obeyed, (3) comfort criteria (e.g. short term aspects such as jerkiness, and long-term aspects such as travel time and traffic jams), and (4) eco-driving compliance. Thus, the criteria can be used to control which situations are instantiated in the dream state.

6.3 Optimal control

Optimal Control (OC) is a broad range of control-theoretic approaches to find optimal control policies to steer a dynamical system from an initial state to a final goal state while optimizing some performance criterion and subject to trajectory constraints.

In cognitive science, it has been acknowledged that Optimal Control well describes human movements, e.g., [21]–[26]. How movement is optimized is actually unclear: there are explanations stating that for movement control that is critical for survival, humans and animals learn to minimize motor noise and maximize movement speed. It was also shown that a particular form of OC (Direct OC) is equivalent to reinforcement learning [27]. We used OC successfully in the driving domain and it turned out to fit fairly well to human trajectories [3], [5], [11], [28]–[32], etc.

For Dreams4Cars, Optimal Control is relevant because it is a method to derive optimal goal-directed actions. This works particularly well for low level strategies such as simple manoeuvres like a lane change, overtake, etc. (see section 7).

The fundamental structure of an OC problem is as follows. The system to be controlled (the "plant") is a dynamical system described as a set of Ordinary Differential Algebraic Equations (or equivalent) such as:

$$\dot{x} = f(x, u) \tag{3}$$

where x is the state vector and u the control input vector (the aim being to find u). The plant dynamic model $f(.)$ in traditional OC is typically an analytical model. There are however examples where the plant dynamics are learnt [23]. Deploying OC on learned forward models is already a simple form of dreaming, in the sense that optimized inverse models are synthetized for the goal at hand given learnt forward models.

The goal is typically defined by a desired final state (5) where the initial state (4) is the current state, meaning that the problem is to find a way to reach the final state from the current situation:

$$x(0) = x_0 \tag{4}$$

$$x(T) = x_T \tag{5}$$

Optimality criteria that work well for human motion are combinations of minimum time and minimum jerk criteria (hence capturing the inherent trade-off between speed and accuracy that is typical in human movement[13]).

[13] Human motor noise is greatly due to neuron noise and is proportional to control signal. Hence minimizing the power of the control signal (which is achieved by minimizing e.g. the square jerk) is a way to minimize also the power of the input noise to the plant.

$$Minimize_{j(t)}\, J = \int_0^T j(t)^T\, dt + w_T T \quad (6)$$

where $j(t)$ is the longitudinal or lateral jerk (derivative of acceleration), T the movement time and w_T a weight to set the trade-off between minimum jerk and minimum time criteria [3], [28]–[30].

Finally, constraints in the state vector trajectory may be imposed such as:

$$g(x,t) < 0 \quad (7)$$

These constraints may be used to impose conditions such as remaining in the lane, remaining within the limits of tire adherence, avoiding space-time positions that are reserved for other traffic (including avoiding of being in the same place and time of another object).

Depending on the model of the plant, there are many methods to solve OC problems. If the plant is a kinematic linear model, closed form solutions exist. These have been used to form “motor primitives” in many of the cited papers and are also at the bottom of the current AdaptIVe agent.

If the plant model is non-linear, such as a learnt plant model, there exist non-linear solvers. The drawback of these solvers is that they are not fast enough to evaluate a full range of motor strategies (hence the motor cortex in Figure 2 could not be computed or could be computed with very coarse resolution). However, this is not a problem for offline use, such as the dreaming mechanism.

One approach that we have found extremely powerful is based in the calculus of variations, combined with computer algebra systems. This approach requires that the plant model can be symbolically differentiated (which is the case with some machine learning methods, for example locally weighted projection regression - LWPR). With symbolic algebra manipulation, what was the main weakness of the vibrational approach (the difficulty in computing the co-state equations) turned out to be a strength (symbolically derived co-state equations greatly help the solution of the problem) [5], [33].

In Dreams4 Cars OC will be used both for generating high-resolution action spaces based on kinematic models and closed form solutions and to generate the same maps with analytical and learnt plant models.

6.4 Exploratory Learning

Perception-Action learning can be approached actively via the randomized identification of perceptual goals within the PA hierarchy in order to generate exploratory training data. Perceptual goals exist at all levels; the subsumptive nature of the hierarchy means that goals and sub-goals are scheduled with increasingly specific content as high-level goals are progressively grounded through the hierarchy (e.g. the high-level intention `drive to work' involves the execution of sub-tasks with correspondingly lower-level perceptual goals e.g. keeping the lane).

This can apply across different mechanisms –in particular, across the implicit symbolic/sub-symbolic divide imposed in the Dreams4Cars architecture (logical reasoning applies only at the top levels of the P-A hierarchy with respect to discrete, highway-code relevant configuration changes [e.g. lane changes]– changes in metric relations, on the other hand [e.g. within-lane changes in car proximity] are implicitly sub-symbolic, and dealt with via optimized motor primitives).

At higher levels, exploratory PA ‘motor babbling’ thus occurs via logically-constrained top-down randomised variable instantiation (predicatization serves as a common interface between hierarchical levels; both low-level sensor-data and higher-level relational concepts can be expressed as predicates of appropriate arity). The PA motor babbling process thus complements goal-directed exploration –i.e. exploration of different strategies to achieve a given high-level goal. New learning is thus initiated when, for example, OC is required by top-down variable instantiation to solve a specific trajectory optimisation problem arising from the novel concatenation of scheduled tasks.

Variable instantiation for exploratory learning will initially be deployed via declarative fuzzy reasoning, with parallel research being undertaken to implement this approach fully in neuro (i.e. with the logic constituting the top layers of the Dreams4Cars neural architecture and top-down instantiation conducted via the usual neural feedback mechanisms). In this case, the logic levels would be fixed; a key ambition is to have the entire neural system fully trainable (logic included); however certain fundamental theoretical issues need to resolve in order to fully achieve this. The exploratory driver will thus either derive from top-level predicates generated via declarative reasoning or else from the output of selected neurons in an integrated neural system.

The Dreams4Cars architecture will enforce compatibility across code bases and code types (in particular, declarative and non-declarative code). The combination of complied and non-compiled code will be initially handled via a simple file based interface (which is suitable given the low-bandwidth of high-level predicate declarations/variable instantiations; high-level fuzzy deduction naturally operates at a lower temporal resolution than the low-level trajectory optimisation). In a fully neuralised implementation, no such inter-mechanism interfacing is required.

Logical predicatization and logical variable description will be designed to mirror the parametric interface to OpenDS described in Section 4, enabling logical variable instantiation to interface with & operate within the cloud environment implementing the OpenDS simulator.

6.5 Action discovery

Action discovery could be defined as the driving system having the capacity to learn new actions that have not been explicitly coded into the system. This problem is an area of active research for a number of groups and is a different kind of problem to the learning that is done in deep neural networks. In this project, we aim to leverage our (Sheffield University) understanding and models of action selection and reinforcement learning in the vertebrate brain. Progress could be made by marrying together reinforcement learning with other forms of artificial intelligence learning.

We will explore the possibility of enabling action discovery by using intrinsic reward. We will attempt to create a system that models the future state of the environment in response to its actions, then compares the real result to its prediction. The intrinsic reward would be delivered in direct proportion to the difference between the model and the reality, thus motivating the system to repeat actions that give rise to unexpected consequences.

In a separate sense, action discovery could be conceived as being similar to learning sequences of actions that are executed smoothly, one after the other. For example, an overtaking manoeuvre is a sequence of change-lane, accelerate, change-lane back again. If the system could learn to execute these manoeuvres in the correct order and in the correct context, then we could reasonably say that the system had learned “overtake” as a distinct action.

This conception of action discovery has a bearing on the system architecture in so far as a history of recently completed actions must be stored and used as input to the selection of the current action.

7 Example

We give here one example to clarify how the Dreams4Cars system should work for learning new behaviours. The example deals with the discovery and learning of overtake manoeuvres on two-lanes roads with traffic in both directions. This means that that the system should learn to find a safe gap and execute an optimized sequence of actions: 1) left lane change, 2) acceleration, 3) right lane change.

Figure 14 shows how the AdaptIVe system (which is the baseline for Dreams4Cars) operates in this case. As said (section 3.2) AdaptIVe uses part of the Dreams4Cars architecture: a layered control architecture with two layers and an action selection mechanism operating on a two-dimensional action space ('motor cortex').

The layered control combines longitudinal motor primitives for speed adaptation with lateral motor primitives for adaptation of the position in the road. At the second (and last) level of the architecture the system uses the motor primitives to produce behaviours such as lane following or lane changes with simultaneous achievement of any uniform speed at any point (including zero speed at a stop line –stop behaviour– or same speed of a leading vehicle as some time gap –car following–). The system does not have higher levels of control and, thus, does not conceive sequences of actions. For example, in a situation like Figure 14, the system might conceive actions such as "keep the lane and follow the car in front" (*a*) or "change lane and get a higher desired speed" (*b*). When travelling on a road with multiple lanes in the same direction, the system will change lane whenever it finds a vehicle that is somewhat slower than its target speed (if the target lane is not occupied by another vehicle). With these two layers, the system may overtake other vehicles but this is an emergent behaviour, resulting from changing to lanes that allow to travel at a desired speed (just like cars switching to fastest lanes in a motorway). The system will change to the right lane whenever this is a as fast as the current lane; hence the system might give then impression of making overtake manoeuvres but the system actually is not planning a sequence of actions.

The system will never overtake on a two-lane road with traffic in both directions. Figure 14, bottom, shows a schematic representation of the action space in this case. The car coming from the opposite direction inhibits all actions ending in the left lane because the system does not make any plan after the lane change: action *b* for the system means changing to the left lane and remaining there, hence colliding sooner or later with the vehicle arriving form the opposite direction. Instead, if the system plans to remaining in the lane, the vehicle ahead will inhibit actions leading to rear end collision so that the system will select action *a*, which is following the front car, even if overtake is possible because there is sufficient gap to manoeuvre.

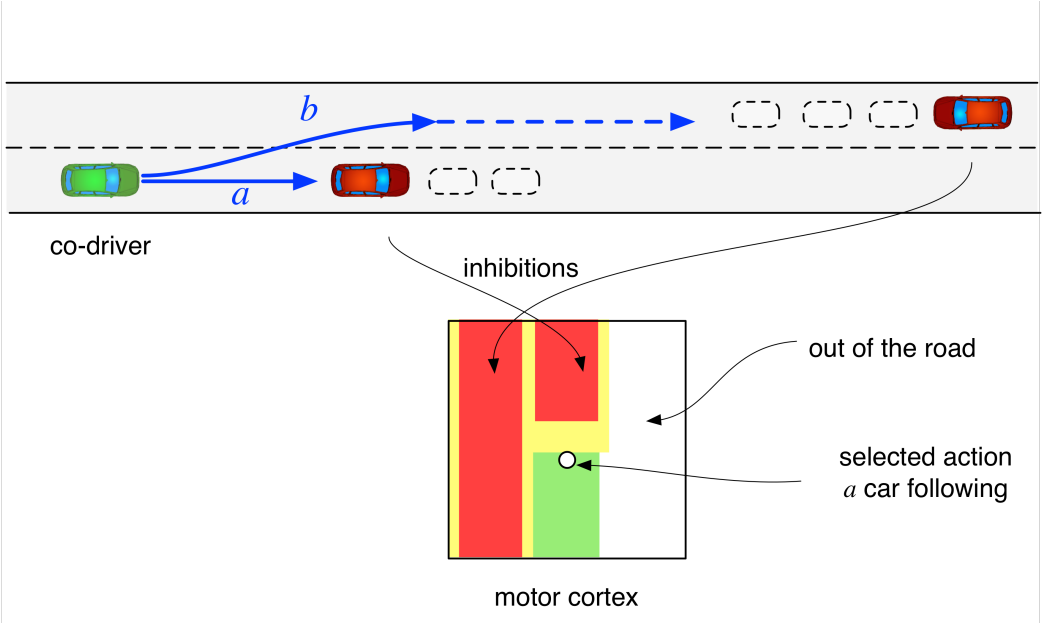


Figure 20: A system that knows only lane change behaviours will never overtake in a two-lane road with traffic in both directions (see text).

Overtake manoeuvres are particularly dangerous manoeuvres. The safe execution of such manoeuvres requires conceiving a sequence of actions, with return to the original lane, and a correct evaluation of the future dynamics of the vehicles.

It is true that one might restrict the operation of automated vehicle to never overtake in two lane roads (this is what AdaptIVe is). Alternatively, one might extend the layered architecture of AdaptIVe with another layer that plans sequences of actions. However, there are many variations in the configuration shown in Figure 14 (position and speed and number of vehicles, actual geometry and condition of the road etc.). Even if the example is so simple, there may be variations that are not considered by the designer when developing the algorithms so that developing and testing the function may require significant work.

How could the Dreams4Cars technology, in this simple example situation, discover and learn overtake behaviours? And what is the final result of learning?

Let us suppose that the system has already developed motor primitives and behaviours like the Adaptive system. Let us call the motor primitives “level 0” and the lane change and longitudinal following behaviour “level 1”. The system is now trying to develop behaviours for a new level, which is “level 2”.

Let the target perceptual goal for level 2 be: “the green car has to be in front of the now-leading red car”. In trying to achieve this goal, the system begins motor babbling at level 2, testing various combinations of the behaviours it already knows at level 1, such as various combinations of lane change and acceleration/deceleration. At this level of abstraction (level 2), the system only conceives sequences of actions, among which the correct sequence 1) left lane change, 2) acceleration, 3) right lane change. However, the parameters for each of these actions (such as how much to accelerate, where exactly to position in the lane, how quickly to change lanes, etc.) are not specified here. It will be responsibility of levels 0 and 1 to optimize the parameters. Hence, when babbling at level 2, the system is actually defining an optimal control problem resulting by the chaining of three sub problems.

Let us assume that we use (in this example) optimal control, using the learnt forward models for the dynamics of the controlled plant. Optimal control provides a synthesis (i.e. the optimal policy) to steer the plant to the desired state and is somewhat equivalent to reusing the forward emulator for infinite simulations until the optimal solution is found (Optimal Control on learnt dynamics is equivalent to reinforcement learning in dreaming states).

Let us suppose that a successful sequence of optimally parametrized action $b'+b''+b'''$ is discovered. At this point the sequence is no longer colliding with the vehicle coming from the opposite direction. Hence the motor space turns to the representation of Figure 15, bottom. In particular, the vehicle coming in the opposite direction is no longer inhibiting the lane change actions and the system will thus choose to change the lane (as the beginning of the sequence $b'+b''+b'''$).

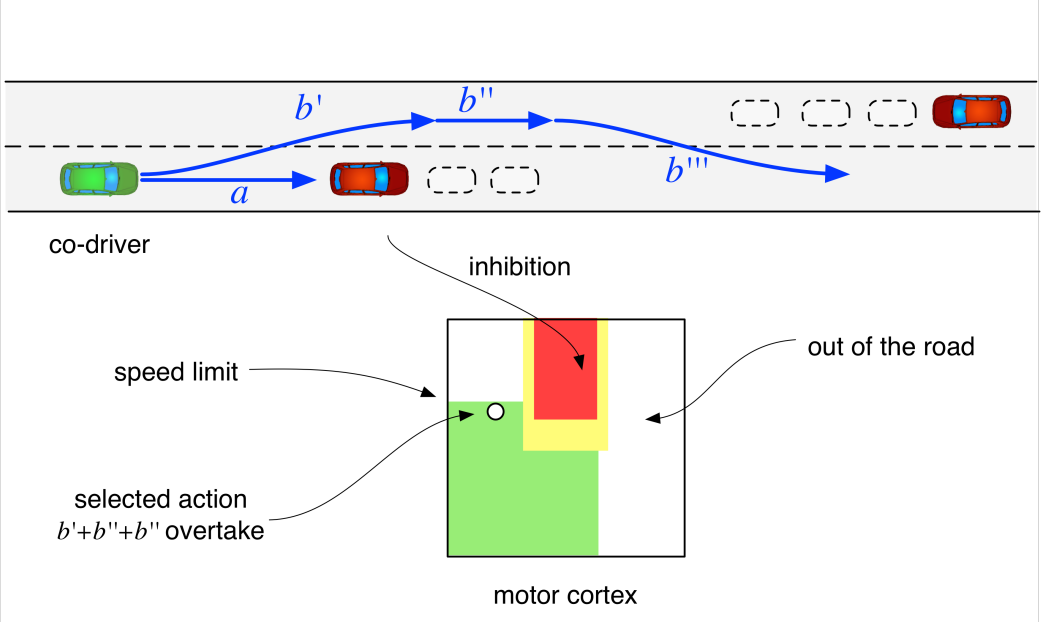


Figure 21: A system that knows only lane change behaviours will never overtake in a two-lane road with traffic in both directions (see text).

Once a successful action is discovered, mechanisms such as motivated learning may be activated. The system will thus simulate many scenarios similar to Figure 15, with variations in the parameters (e.g., the distance of the incoming vehicle may be different). In this way, the system should learn mapping from the perceptual space (figure 15, top) to the motor space (figure 15, bottom) so that the recognition that the overtake action is possible will happen on the fly (if the incoming vehicle is too close it will still inhibit the lane change and no overtake manoeuvre will, begin).

Simultaneous with learning the above mapping, the system will also learn that certain sequences of actions are possible/impossible given the context, hence not only learning the instantaneous control, but also learning the whole sequence of actions.

8 Bibliographical References

[1] P. Cisek, "Cortical mechanisms of action selection: the affordance competition hypothesis," *PhilosTransRSocLond B Biol Sci*, vol. 362, no. 1485, pp. 1585–1599, 2007.

[2] M. Bojarski *et al.*, "End to end learning for self-driving cars," *ArXiv Prepr. ArXiv160407316*, 2016.

[3] M. Da Lio *et al.*, "Artificial Co-Drivers as a Universal Enabling Technology for Future Intelligent Vehicles and Transportation Systems," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 1, pp. 244–263, 2015.

[4] R. Bogacz and K. Gurney, "The basal ganglia and cortex implement optimal decision making between alternative actions.," *Neural Comput.*, vol. 19, no. 2, pp. 442–477, Feb. 2007.

[5] E. Bertolazzi, F. Biral, and M. Da Lio, "Symbolic-numeric efficient solution of optimal control problems for multibody systems," *J. Comput. Appl. Math.*, vol. 185, no. 2, pp. 404–421, 2006.

[6] A. Dosovitskiy and T. Brox, "Inverting visual representations with convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4829–4837.

[7] A. Dosovitskiy, J. Springenberg, M. Tatarchenko, and T. Brox, "Learning to generate chairs, tables and cars with convolutional networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2016.

[8] M. Bojarski *et al.*, "Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car," *ArXiv Prepr. ArXiv170407911*, 2017.

[9] D. Betaille and R. Toledo-Moreo, "Creating Enhanced Maps for Lane-Level Vehicle Navigation," *IEEE Trans. Intell. Transp. Syst.*, vol. 11, no. 4, pp. 786–798, Dec. 2010.

[10] P. Bender, J. Ziegler, and C. Stiller, "Lanelets: Efficient map representation for autonomous driving," in *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*, 2014, pp. 420–425.

[11] V. Cossalter, M. Da Lio, R. Lot, and L. Fabbri, "A general method for the evaluation of vehicle manoeuvrability with special emphasis on motorcycles," *Veh. Syst. Dyn.*, vol. 31, no. 2, pp. 113–135, 1999.

[12] C. Rauch, E. Berghöfer, T. Köhler, and F. Kirchner, "Comparison of Sensor-Feedback Prediction Methods for Robust Behavior Execution," in *KI 2013: Advances in Artificial Intelligence*, Springer, 2013, pp. 200–211.

[13] R. Math, A. Mahr, M. M. Moniri, and C. Müller, "OpenDS: A new open-source driving simulator for research," in *Adjunct Proceedings of the AutomotiveUI'12 Conference*, Portsmouth, NH, USA, 2012.

[14] K. Maddock, "Vehicle Simulation with Bullet. https://docs.google.com/document/d/18edpOwtGgCwNyvakS78jxMajCuezotCU_0iezcwiFQc/edit?pref=2&pli=1." 16-Aug-2010.

[15] "CityEngine Quick Start Guide," *http://cehelp.esri.com/help/index.jsp?topic=/com.procedural.cityengine.help/html/quickstart/overview.html.* .

[16] H. Svensson, S. Thill, and T. Ziemke, "Dreaming of electric sheep? Exploring the functions of dream-like mechanisms in the development of mental imagery simulations," *Adapt. Behav.*, vol. 21, no. 4, pp. 222–238, 2013.

[17] H. Svensson and S. Thill, "Beyond bodily anticipation: Internal simulations in social interaction," *Cogn. Syst. Res.*, vol. 40, pp. 161–171, 2016.

[18] R. Cotterill, *Enchanted looms conscious networks in brains and computers /*. Cambridge, UK ; Cambridge University Press, 1998.

[19] G. Hesslow, "The current status of the simulation theory of cognition.," *Brain Res.*, vol. 1428, pp. 71–9, Jan. 2012.

[20] H. Svensson, *Simulations*. Linköping: Linköping University Electronic Press, 2013.

[21] S. G. Khan, G. Herrmann, F. L. Lewis, T. Pipe, and C. Melhuish, "Reinforcement learning and optimal adaptive control: An overview and implementation examples," *Annu. Rev. Control*, vol. 36, no. 1, pp. 42–59, 2012.

[22] D. Liu and E. Todorov, “Evidence for the flexible sensorimotor strategies predicted by optimal feedback control.,” *J. Neurosci. Off. J. Soc. Neurosci.*, vol. 27, no. 35, pp. 9354–68, Aug. 2007.

[23] D. Mitrovic, S. Klanke, and S. Vijayakumar, “Adaptive Optimal Feedback Control with Learned Internal Dynamics Models,” *Robotics*, vol. 264, pp. 65–84, 2010.

[24] A. J. Nagengast, D. a Braun, and D. M. Wolpert, “Optimal control predicts human performance on objects with internal degrees of freedom.,” *PLoS Comput. Biol.*, vol. 5, no. 6, p. e1000419, Jun. 2009.

[25] P. Viviani and T. Flash, “Minimum-jerk, two-thirds power law, and isochrony: converging approaches to movement planning.,” *J. Exp. Psychol. Hum. Percept. Perform.*, vol. 21, no. 1, pp. 32–53, Mar. 1995.

[26] C. M. Harris, “Biomimetics of human movement: functional or aesthetic?,” *Bioinspir. Biomim.*, vol. 4, no. 3, p. 33001, Sep. 2009.

[27] R. S. Sutton, A. G. Barto, and R. J. Williams, “Reinforcement learning is direct adaptive optimal control,” *IEEE Control Syst. Mag.*, vol. 12, no. 2, pp. 19–22, 1992.

[28] M. Da Lio, A. Mazzalai, K. Gurney, and A. Saroldi, “Biologically Guided Driver Modeling: the Stop Behavior of Human Car Drivers,” *IEEE Trans. Intell. Transp. Syst.*, vol. submitted.

[29] M. Da Lio, A. Mazzalai, and M. Darin, “Cooperative Intersection Support System Based on Mirroring Mechanisms Enacted by Bio-Inspired Layered Control Architecture,” *IEEE Trans. Intell. Transp. Syst.*, vol. submitted.

[30] P. Bosetti, M. Da Lio, and A. Saroldi, “On Curve Negotiation: From Driver Support to Automation,” *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 4, pp. 2082–2093, 2015.

[31] F. Biral, E. Bertolazzi, and M. Da Lio, “The Optimal Manoeuvre,” in *Modelling, Simulation and Control of Two-Wheeled Vehicles*, Wiley, 2014, pp. 119–154.

[32] F. Biral, M. da Lio, R. Lot, and R. Sartori, “An intelligent curve warning system for powered two wheel vehicles,” *Eur. Transp. Res. Rev.*, vol. 2, no. 3, pp. 147–156, 2010.

[33] E. Bertolazzi, F. Biral, and M. Da Lio, “Symbolic-numeric indirect method for solving optimal control problems for large multibody systems: The time-optimal racing vehicle example,” *Multibody Syst. Dyn.*, vol. 13, no. 2, pp. 233–252, 2005.

[34] Synergy Research Group, “Amazon Cloud Growth is Hardly Hampered by the Chasing Pack,” *https://www.srgresearch.com/articles/amazon-cloud-growth-hardly-hampered-chasing-pack*. .

[35] E. Knaser and D. Ahuja, “In-Depth Assessment of Amazon Web Services,” *https://www.gartner.com/doc/3371747*, 14-Jul-2016. .

[36] K. Hilgendorf and D. Ahuja, “In-Depth Assessment of Microsoft Azure IaaS,” *https://www.gartner.com/doc/3371748*, 14-Jul-2016. .

[37] D. Toombs and D. Ahuja, “In-Depth Assessment of Google Cloud Platform,” *https://www.gartner.com/doc/3377519*, 13-Jul-2016. .

[38] E. Knaser, “Evaluation Criteria for Cloud Infrastructure as a Service,” *https://www.gartner.com/doc/3322017*, 18-May-2016. .

9 Appendix

9.1 Comparison of Major Cloud Service Providers

In this section we investigate on the hardware requirements of the proposed cloud environment and compare major cloud service providers that qualify for hosting the simulation environment as a cloud service (which is a possible follow up product of the project).

According to Synergy Research Group [34], the most influential players in the public cloud provider market by end of March 2017 are Amazon, Microsoft, IBM, and Google. While the worldwide market share of market leader Amazon Web Services (AWS) is holding steady at 33%, Microsoft, IBM, and Google are gaining ground at the expense of smaller players (c.f. Figure 22). Microsoft and Google achieved annual growth rates of more than 80% (AWS <50%); however, AWS revenues are still comfortably bigger than the other three combined. Continuing this trend, Google and Microsoft might catch up with Amazon very soon. Today, the market share of these three operators is larger than those of all the other (200+) providers together.

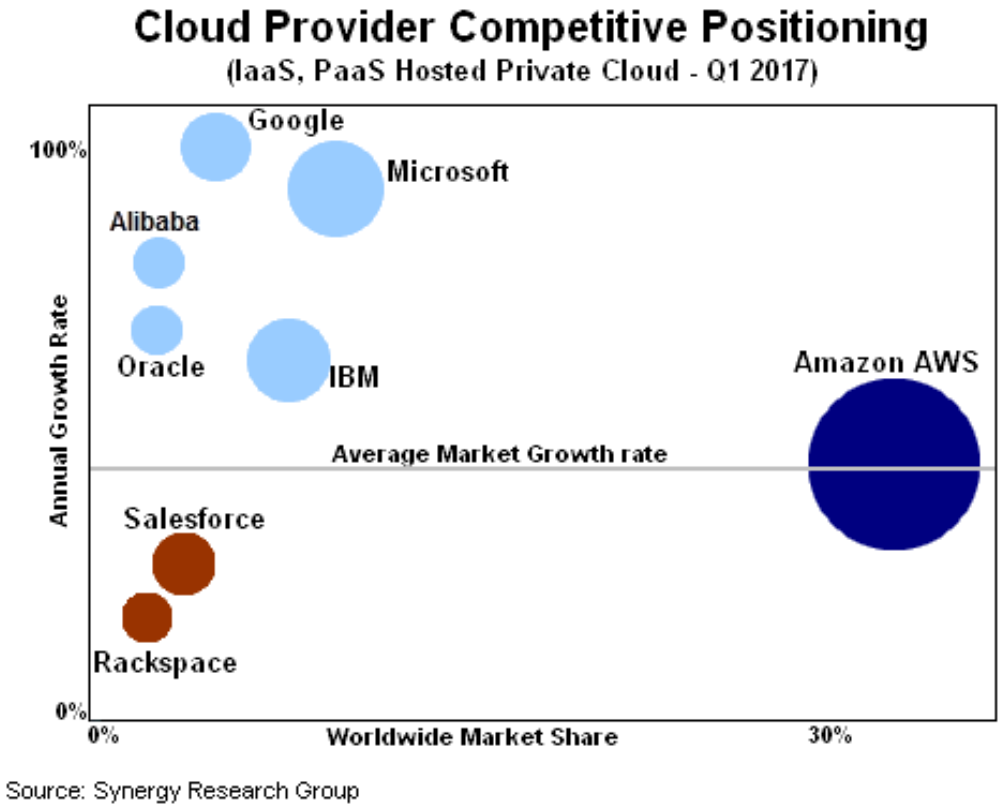


Figure 22: Cloud Provider Growth Rate and Market Share

While IBM continues to lead in hosted private cloud; Amazon, Microsoft, and Google are the lead providers in IaaS/PaaS. In particular AWS dominates the IaaS market; while Microsoft's Windows Azure slightly leads the PaaS market; Google Cloud Platform finishes up in third place in both categories.

With regard to the proposed cloud environment of this project, we narrow down our view to the top three public cloud providers in PaaS and rely on the respective in-depth assessments published in May 2016 by Gartner, Inc., an American research and advisory firm providing information technology related insight for IT and other business leaders located across the world [35], [36], [37]. Research and scores are based on the *Evaluation Criteria for Cloud Infrastructure as a Service* [38] which is made up of 234 criteria items. The 234 criteria items are organized into four technical categories (Compute, Networking, Storage, Security and Access) and four non-technical categories (Service Offerings, Management and DevOps, Service and Support Levels, Price and Billing). Within each of the above categories, criteria have been organised into Required, Preferred

and Optional sections. Table 6 depicts the overall score where Amazon scored highest in all sections followed by Microsoft and Google.

	Required	Preferred	Optional
Amazon Web Services	92%	71%	61%
Microsoft Azure	88%	57%	47%
Google Cloud Platform	70%	41%	24%

Table 6: Score of different cloud providers based on 234 criteria items.

Regarding a rapidly growing choice of products, service improvements in short intervals and prices dropping on an almost daily basis, we are aware that long-term predictions are not possible and figures like the aforementioned ones were most probably outdated before they had been published. In addition to the overall results, which already provide a good indication about the quality of service of these three providers, we investigate on the most critical features concerning the proposed simulation environment, i.e. the service categories compute, storage, networking, and pricing structure. For assessment and better comparison of the different platforms we define the following reference specification for computing hardware of which we know that the proposed driving simulation software is showing good performance.

- Intel i7 processor (8 cores, 2.5 GHz)
- NVIDIA GeForce GTX 970 graphics card[14] (1664 CUDA cores, 4 GB video memory)
- 16 GB RAM
- 30 GB SSD (operating system and simulation environment)
- 100 GB HDD “cold storage” (log data)
- Operating system: Windows, Mac OS X, Linux
- Java Runtime Environment (version 8)
- Relational data base
- Static IP address

9.1.1 Amazon Web Services

Amazon Web Services offers a very comprehensive pay-per-use cloud service consisting of many different system components that could be used to set up almost any configuration. The Elastic Compute Cloud[15] (EC2), which provides scalable computing on demand, allows the user to choose from 57 different instance types varying in CPU power, memory size, network performance, and presence of one or more GPUs. The instance types are grouped into general purpose, GPU enabled, compute optimized, memory optimized, and storage optimized instances where each group consists of instances ranging from low-performance instances of the previous generation to high-performance instances of the latest generation. In addition to the computing capability, the user can add any amount of persistent storage, which is called Amazon Elastic Block Store (EBS), and choose from 63,508 Amazon Machine Images (AMI). An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch an instance. Provided operating systems are several 32-bit and 64-bit versions of Amazon Linux, Cent OS, Debian, Fedora, Gentoo, OpenSUSE, Red Hat, SUSE Linux, Ubuntu, and Windows.

Out of all 57 instance types, Amazon provides only two instance types that are optimized for graphics-intensive applications on the one hand and meet the requirements of the reference specification on the other

[14] A graphics card is required to render the simulation screen (e.g. video for validation) and for computation of several image-based sensors (lidar, radar, etc.)

[15] Amazon Elastic Computing Cloud: https://aws.amazon.com/ec2

hand. Table 7 compares both instance types concerning number of GPUs, (virtual) CPUs, (local) memory and (local) storage.

Model	GPUs	vCPU	Mem (GiB)	SSD Storage (GB)
g2.2xlarge	1	8	15	1 x 60
g2.8xlarge	4	32	60	2 x 120

Table 7: Amazon EC2 instances for graphics-intensive applications.

Both instance types feature one or four high-performance NVIDIA GPUs (each with 1,536 CUDA cores and 4GB of video memory) as well as high-frequency Intel Xeon E5-2670 processors (8 cores, 2.6 GHz). In order to meet the specification requirements, we add 30 GB SSD storage and 100 GB HDD storage (EBS) for logging. If that amount is not sufficient it might be worth to consider Amazon's low-cost cloud storage service Amazon Glacier which is up to 75% cheaper; however, access times vary from a few minutes to several hours and additional data retrieval fees apply. The most suitable operating system is the 64-bit version of Microsoft Windows Server 2016 which comes at a slightly higher cost compared to Linux instances. Once the server has been set up, the user can install further software (such as the Java Runtime Environment) on his own using a remote connection and set up a static IP address to his cloud using Elastic IP Address.

In the following we calculate the expected cost per 24 hours where we assume to use the closest Availability Zone (Frankfurt am Main, Germany) in order to minimize latency. Pricing is per instance-hour consumed for each instance, from the time an instance is launched until it is stopped. Each partial instance-hour consumed will be billed as a full hour. Table 8 shows the cost of operating a single-GPU instance for 24 hours whereas Table 9 shows the cost of an instance with four GPUs.

Item	Cost per unit	Unit	Cost per 24 hours
Instance g2.2xlarge (incl. Windows OS)	$ 0.906	hour	$ 21.744
Additional cost for EBS-optimized instance	$ 0.05	hour	$ 1.20
30 GB SSD Elastic Block Store	$ 0.119	GB/month	$ 0.119
100 GB HDD Elastic Block Store	$ 0.03	GB/month	$ 0.10
1 TB data transfer per month (out only)	$ 0.09	GB	$ 3.00
Static IP Address	$ 0.00	hour	$ 0.00
			$ 26.16

Table 8: Cost per 24 hours of 1-GPU instance operation (Amazon EC2).

Item	Cost per unit	Unit	Cost per 24 hours
Instance g2.8xlarge (incl. Windows OS)	$ 3.366	hour	$ 80.784
30 GB SSD Elastic Block Store	$ 0.119	GB/month	$ 0.119
100 GB HDD Elastic Block Store	$ 0.03	GB/month	$ 0.10
1 TB data transfer per month (out only)	$ 0.09	GB	$ 3.00
Static IP Address	$ 0.00	hour	$ 0.00
			$ 84.00

Table 9: Cost per 24 hours of 4-GPU instance operation (Amazon EC2).

9.1.2 Microsoft Azure

Microsoft Azure offers services that are very similar to Amazon's services: Azure Virtual Machines[16] allows the user to choose from 49 instance types which are grouped into the same categories as the instance types of Amazon's EC2: general purpose, GPU enabled, compute optimized, memory optimized, storage optimized instances – plus a group named "high-performance computing" containing six additional instances. Like Amazon instances, Azure instances are constantly getting replaced by the latest generation of computing hardware, whereas older generations will still be offered at a discount. In contrast to AWS, Azure provides a decent amount of basic RAM and storage for all packets. Additionally, the user can add any amount of scalable cloud storage from the so-called Data Lake Store (DLS). For bigger amounts of data with low access rates or additional data redundancy options, Azure Storage might be an economic alternative. The user can choose from several ready-to-use operating systems such as Cent OS, Ubuntu, Red Hat, R-Server, SUSE Linux, and Windows.

[16] Microsoft Azure: https://azure.microsoft.com

In total, Azure provides three GPU-enabled instance types that meet the requirements of the reference specification. Table 10 compares the instance types concerning number of GPUs, (virtual) CPUs, (local) memory and (local) storage.

Model	GPUs	vCPU	Mem (GiB)	SSD Storage (GB)
NV6	1X M60	6	56	340
NV12	2X M60	12	112	680
NV24	4X M60	24	224	1,440

Table 10: Microsoft Azure instances for graphics-intensive applications.

All GPU-enabled instance types feature one, two, or four high-performance NVIDIA TESLA-M60 GPUs (each with 4,096 CUDA cores and 16 GB of video memory). In order to meet the specification requirements, we do not need to add any additional storage, as the smallest instance type already comprises of 340 GB local SSD storage. The most suitable operating system is the 64-bit version of Microsoft Windows Server which comes at a slightly higher cost compared to Linux instances. Once the server has been set up, the user can install further software (such as the Java Runtime Environment) and set up a static IP address at additional cost.

Table 11, Table 12 and Table 13 show the cost of operating a single-GPU, a double-GPU, or quadruple-GPU instance of Microsoft Azure Virtual Machines, respectively. The closest Region that provides GPU-enabled instances is Europe North (Ireland). Pricing is per instance-minute consumed for each instance, from the time an instance is launched until it is stopped.

Item	Cost per unit	Unit	Cost per 24 hours
Instance NV6 (incl. Windows OS)	$ 1.46	hour	$ 35.04
1 TB data transfer per month (out only)	$ 0.087	GB	$ 2.90
Static IP Address	$ 0.004	hour	$ 0.096
			$ 38.04

Table 11: Cost per 24 hours of 1-GPU instance operation (Microsoft Azure).

Item	Cost per unit	Unit	Cost per 24 hours
Instance NV12 (incl. Windows OS)	$ 2.92	hour	$ 70.08
1 TB data transfer per month (out only)	$ 0.087	GB	$ 2.90
Static IP Address	$ 0.004	hour	$ 0.096
			$ 73.08

Table 12: Cost per 24 hours of 2-GPU instance operation (Microsoft Azure).

Item	Cost per unit	Unit	Cost per 24 hours
Instance NV24 (incl. Windows OS)	$ 5.83	hour	$ 139.92
1 TB data transfer per month (out only)	$ 0.087	GB	$ 2.90
Static IP Address	$ 0.004	hour	$ 0.096
			$ 142.92

Table 13: Cost per 24 hours of 4-GPU instance operation (Microsoft Azure).

9.1.3 Google Cloud Platform

Google's service for scalable computing on demand is called Google Compute Engine[17]. In contrast to Amazon EC2 and Microsoft Azure Virtual Machines, this service does not only provide a number of pre-defined instance types, but rather allows the customer to define his own so-called machine types. All 21 pre-defined machine types are cheaper compared to the equivalent custom machine type; however, it could pay off to build your own type if choosing the next bigger pre-defined type could be avoided. Unlike competitive cloud providers, Google does not provide GPU-enabled machine types by default. Instead, Google Compute Engine GPUs – which is still in beta release – allows adding up to 4 NVIDIA Tesla K80 GPUs (4,992 NVIDIA CUDA cores, 24 GB video memory) to any machine type. In order to compare Google's service to Amazon's and Microsoft's, we choose the smallest pre-defined machine type that meets the requirements of the reference specification and furthermore build a custom machine type according to the exact requirements. Table 14 compares both machine types concerning number of GPUs, (virtual) CPUs, (local) memory and (local) storage.

Model	GPUs	vCPU	Mem (GiB)	SSD Storage (GB)
n1-standard-8	1	8	30	0
CUSTOM	1	8	15	0

Table 14: Google Compute Engine instances for graphics-intensive applications.

In addition to the computational capabilities, Google Computing Engine provides 30 GB of HDD persistent disk storage per month free of charge; additional storage can be added. Furthermore, premium images containing one of the following operating systems are available at extra charge: Debian, CentOS, CoreOS, SUSE, Ubuntu, Red Hat, FreeBSD, or Windows Server 2008 R2, 2012 R2, and 2016.

In order to meet the specification requirements, we add 30 GB SSD storage as well as 70 GB HDD storage to the free quota and select Microsoft Windows Server 2016 64-bit as operating system. Once the server has been set up, the Java Runtime Environment can be installed by the user and a static IP address can be set up at no extra charge.

Table 15 and Table 16 show the cost of operating the most suitable pre-defined and custom machine type using a single GPU, respectively, where prices are valid for a cloud located in Belgium. Instance uptime is rounded up to the nearest minute; however, Google Compute Engine bills for a minimum of 10 minutes of usage.

[17] Google Compute Engine: http://console.cloud.google.com/compute

Item	Cost per unit	Unit	Cost per 24 hours
Instance n1-standard-8	$ 0.4184	hour	$ 10.0416
GPU (NVIDIA Tesla K80)	$ 1.54	hour	$ 36.96
30 GB SSD Elastic Block Store	$ 0.17	GB/month	$ 0.17
70 GB HDD Elastic Block Store	$ 0.04	GB/month	$ 0.0933
Windows Server image	$ 0.32	hour	$ 7.68
1 TB data transfer per month (out only)	$ 0.12	GB	$ 4.00
Static IP Address	$ 0.00	hour	$ 0.00
			$ 58.94

Table 15: Cost per 24 hours of pre-defined 1-GPU instance operation (Google Compute Engine).

Item	Cost per unit	Unit	Cost per 24 hours
Custom instance 8 vCPUs	$ 0.291912	hour	$ 7.0059
Custom instance 16 GB memory	$ 0.078272	hour	$ 1.8785
GPU (NVIDIA Tesla K80)	$ 1.54	hour	$ 36.96
30 GB SSD Elastic Block Store	$ 0.17	GB/month	$ 0.17
70 GB HDD Elastic Block Store	$ 0.04	GB/month	$ 0.093
Windows Server image	$ 0.32	hour	$ 7.68
1 TB data transfer per month (out only)	$ 0.12	GB	$ 4.00
Static IP Address	$ 0.00	hour	$ 0.00
			$ 57.79

Table 16: Cost per 24 hours of custom 1-GPU instance operation (Google Compute Engine).

Comparing the prices of the pre-defined machine instance and the custom instance reveals no big difference. On the one hand, the custom instance is $ 1.15 cheaper per 24 hours; on the other hand, the pre-defined machine type is equipped with the double amount of memory. Increasing the number of GPUs is possible to a maximum of 4, while the price of the GPU-hour is increasing proportionally. Table 17 shows the cost of operating the aforementioned setup for 24 hours with one, two and four GPUs.

Item	Cost per 24 hours
Instance n1-standard-8 with 1 GPU	$ 58.94
Instance n1-standard-8 with 2 GPU	$ 95.90
Instance n1-standard-8 with 4 GPU	$ 169.82

Table 17: Cost per 24 hours of pre-defined 1-GPU, 2-GPU, and 4-GPU instance operation (Google Compute Engine).

9.1.4 Conclusions

Comparing the prices of operating the same setup with Microsoft Azure and Google Compute Engine shows a considerable difference. No matter what number of GPUs will be used, Google is up to 35% more expensive than competitor Microsoft. Taking a closer look at Amazon reveals the lowest price for the 1-GPU setup; however, one must keep in mind that the smallest Amazon instance type (“g2.2xlarge”) is less powerful than the smallest Microsoft instance type (“NV6”). While the GPU used at Amazon consists of only 1664 CUDA cores and 4 GB video memory, Microsoft (and Google) utilizes GPUs consisting of 4,096 (4,992) CUDA cores and 16 (24) GB video memory. Amazon’s biggest instance type (“g2.8xlarge”) is on the same level with Microsoft’s mid-size instance type (“NV12”) for a slightly higher price, whereas Amazon lacks of instance types comparable to Microsoft’s biggest instance type (“NV24”).

Overall, the cheapest solution which meets the requirements of the reference specification is provided by Amazon (“g2.2xlarge”) using one GPU at $ 26.16 per day. If more computing power is needed, Microsoft provides solutions with two (“NV12”) or four (“NV24”) GPUs at $ 73.08 or $ 149.92 per day, respectively. Google cannot compete.