A Knowledge Base to Dynamically Generate Context-Dependent Software Architectures for Autonomous Vehicles

Philippe Morignot, Guillaume Bresson, Mohamed-Chérif Rahal
Institute VEDECOM, 77, rue des chantiers
78000 Versailles, France
{Philippe.Morignot Guillaume.Bresson Mohamed.Rahal}@vedecom.fr

Sébastien Glaser
IFSTTAR & Institute VEDECOM, 77, rue des chantiers
78000 Versailles, France
Sebastien.Glaser@vedecom.fr

***Abstract*—An autonomous vehicle, towards the goal of being intelligent, must drive in several contexts along its way to destination (e.g., crowded urban streets, straight highways, roundabouts). However, the algorithms (e.g., for perception) running on the automated vehicle in one environment (e.g., in urban zones) are not the same as those running in another one (e.g., on highways). In this paper, we present an approach, based on a knowledge base, to dynamically generate the software architecture of an autonomous vehicle, i.e., make the autonomous vehicle adaptable and context-dependent. Our approach is less CPU power consuming than representing all possible software components inside a large architecture, and switching from one component to another as context changes. Attractive results in simulation are presented, proving the feasibility of the concept.**

Keywords—intelligent vehicle, autonomous driving, context adaptation.

I. Introduction

The dream of the Intelligent Transportation Systems community probably is to build a vehicle which could autonomously and safely drive in every environment encountered on the way to a given destination. For example, a target demonstration of such intelligent vehicle could be to safely drive from *Place de l'Etoile* roundabout in Paris at 5PM, take the *Avenue de la Grande Armée* (urban environment), then turn around *Porte Maillot* roundabout, take the *Périphérique Extérieur* (highway), then take the *A13* (highway), then enter Versailles (urban environment) to reach *Rue des Chantiers* --- this path includes 2 urban environments, 2 roundabouts and 2 highways. Closest demonstrations are the DARPA Urban Challenge won by the Junior car (2007) or international projects such as VIAV (international travel from Parma to Shanghai) by VisLab in 2010, among others.

However, in an intelligent autonomous vehicle, the algorithms running to drive the automated vehicle in one environment are not the same as those running to drive it in others: these algorithms are context-dependent. This comes from the fact that an algorithm is usually based on assumptions, i.e., given specific inputs an algorithm will produce more or less performant outputs, depending on the un/satisfaction of these assumptions. For example, a perception algorithm for lane detection and tracking [13] may exhibit a lane recognition rate which will be high for straight roads, e.g., on highways, and low for curved roads, e.g., in mountains. Assumptions (performance/applicability zones) of an algorithm indeed appear as of crucial importance for intelligent vehicles autonomously driven by computer software.

In this paper, we present an approach to make an autonomous vehicle adaptable to its environment/context. Our approach is based on a knowledge base to off line describe the algorithms (e.g., perception) running inside an automated vehicle, and to dynamically choose on line (i.e., while the automated vehicle is driving) which algorithms to use in the context at hand. A key point of our approach is to consider a software architecture as a graph where nodes are algorithms and vertices are data flows.

Cognitive architectures, i.e., software architectures including both reaction and deliberation, have been studied for long in robotics and Artificial Intelligence, e.g., [1][5][6]. Model-driven engineering has also been used to generate software architectures of robotic systems, e.g. [12], considered as problem solving. But to our knowledge, this is the first time that such cognitive architecture is proposed for autonomous vehicles in the domain of Intelligent Transportation Systems.

This paper is organized as follows: in section II, literature on cognitive architectures and model-based engineering is compared to our knowledge-based approach; In section III, existential software architectures of autonomous vehicles are presented, knowledge-based reasoning is recalled and applied to representation of assumptions of algorithms, in order to exhibit dependency on environment/context; in section IV, an implementation including an inference engine and a proprietary tool for activating software architectures is described, and a control algorithm for testing our implementation on actual data logs is presented; section V discusses our model regarding issues of safety, embedded reasoning and response time. Finally, we sum up our contribution and propose extensions.

II. Related Work

Many autonomous vehicles are designed for a unique context/environment, e.g., [2]. But autonomous driving of vehicles involves driving in every context encountered along the way to destination (see target scenario in section I), in order to reach adaptability.

Cognitive architectures, i.e., software architecture including both reaction and deliberation, have been studied for long in robotics and Artificial Intelligence and are a debated topic. [7] proposes a two-layer software architecture for controlling autonomous robotic agents, based on a "cognitive"' layer including time consuming components (e.g., action/task planning) and a "reactive" layer including fast reactive loops connected to the environment. Each layer is organized as a *blackboard architecture* [6]: A data structure (the "blackboard") is visible/accessible by all *knowledge sources* (internal agents), and knowledge sources react to changes on this structure by bringing knowledge to it, with a *control plan* for choosing which knowledge source actually accesses it in case of conflict. This 2-level architecture has been successfully used to control an indoor mobile robot, acting as a fac totum in offices. A work close in spirit is [10], which proposes a fully parallel software architecture based on Device Profile Web Services to encapsulate a task/action planner as a web service and generate linear scenarios (i.e., a sequence of high-level tasks), controlling a mobile robot with an arm in indoor static environments

But both work, despite including a task/action planner, which is potentially time consuming, are based on the response time of the deliberative components on actually encountered cases, which is paradoxically high (fast action/task planning) when compared to the (low) motion speed of these mobile robots. Unfortunately, this assumption does not hold for an autonomous vehicle driving at speed 130km/h on highways. In addition, these mobile robots run in static environments, which is not the case for an autonomous vehicle driving in urban zones (a dynamic environment).

In contrast, [1] proposes a 3-layer cognitive architecture, composed of a "deliberation" layer (including task/action planning and procedural reasoning), a "functional" layer (choosing which behavior of the agent to activate) and a "control" layer (executing the prescribed behavior). Another three layer architecture is proposed in [5], which is composed of a "deliberator" (containing time-consuming search-based algorithms such as task planning), a "controller" (containing fast feedback control loops) and a "sequencer" (fast selection of the behavior to activate and conditional reaction to unexpected output). As opposed to these bodies of work, if the whole software architecture of ITS may probably fall into a behavior of the previous "controller"s,, the proposed knowledge-based reasoning approach is much faster than time consuming algorithms such as task planning, therefore it could probably be contained in the previous "functional" / "Controller" layers (knowledge based generation of "behaviors") under this view --- see also 2nd point of section V.

[12] proposes a model-driven engineering approach defining a language for expressing functional and non-functional properties of a system and expressing software architecture as a solution resulting from problem solving. But in these authors' view, a software architecture is considered as a solution to a problem, and as such might be subject to time consuming computation, whereas in our approach algorithms encapsulated in components result from an expertise, which is less time consuming than solving a combinatorial problem (except for small sized problems). In addition, our approach is dedicated to autonomous vehicles, in which expertise exists, whereas these authors' approach is more generally dedicated to robotic systems.

III. Model

A. Software architectures

The algorithms inside an autonomous vehicle are usually organized in a software architecture, i.e., a man-designed directed acyclic graph of nodes (algorithms) and vertices (data flow) --- as depicted in Fig. 1 Proprietary tools for graphically representing/designing and running such architectures, i.e., synchronizing data streams, include RTMaps (Real Time Multi-Sensor Advanced Prototyping Software [11]) and ROS (Robotic Operating System). These tools can be used at design time for graphically designing in a user-friendly way such graphs (encapsulating algorithms inside components/nodes and stating data streams as directed vertices between a component output and another component input), or at run time for activating the executable code of the algorithms (nodes) in threads, making them exchange data through ports (vertices), and actually driving an intelligent vehicle with its sensors and actuators through the SLAM / perception / data fusion / path planning / control cycle, common in the ITS domain.

Unfortunately, such software architecture can be designed and run with the above 2 tools but cannot be changed once launched in an autonomous vehicle. The only exception to this is the "Condition" component of the RTMaps proprietary tool [11], which blocks an output port if a condition is not met. That component can be used to feed/starve parts of a software architecture but all parts would still run, consuming CPU power / resources and leading to large software architectures (representing all cases) with a small portion of it being active/fed only. On the opposite, runtime conditions of the autonomous vehicle require to dedicate the whole CPU to active components of reasonably large software architecture, to ensure data flow speed from the autonomous vehicle's sensors to its actuators --- a data delay of e.g. 1s in the output (e.g., the socket sender component at the middle right of Fig. 1) could lead to unsafe manoeuvers of the autonomous vehicle (e.g., on a highway with an autonomous vehicle driving at 130km/h, a 1s delay represents 36.11m).

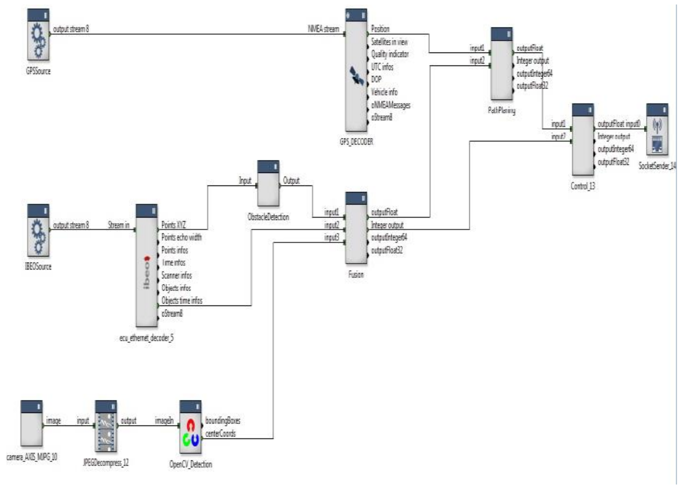

Fig. 1. Visualization of a generic (simplified) software architecture for driving a Zoé autonomous vehicle (vehicle augmented with sensors, effectors and computing power from a product of the Renault company). The lower left branch represents camera acquisition, the middle left one represents a LIDAR sensor acquisition the upper left branch represents a GPS sensor, the middle component represents data fusion, the 2 composenents on the top and middle right represent path planning and control components. The resulting command is sent to the vehicule via a socket with the middle far right component.

B. *Existential software architectures*

Towards this goal, we propose existential software architecture, i.e., software architectures which exist due to context / environment and which can dynamically change as the autonomous vehicle's context changes and evolves. For all context/environment, there exists a software architecture such that this software architecture is adapted to this environment --- since most drivers can drive in any environmental conditions. For example, the target path described in the section I requires 3 software architectures due to the presence of 3 distinct environments / contexts (roundabout, urban zone, highway).

Making intelligent vehicle's software architectures existential, adaptable and context dependent can be performed based on the following key point: these tools represent a specific software architecture in a computer language (an XML file and associated libraries of component's executable in the case of RTMaps [11]), which as such can be the output of another software mechanism. This is the feature which is used in this paper, for context dependent dynamic generation of software architecture of an intelligent vehicle. A software architecture itself is considered as a programming object, which is once generated by a designer but which can later be dynamically re-generated not by a designer but by a computer (another piece of software).

C. *Knowledge base*

For performing this dynamic context-dependent online generation of software architecture for an autonomous vehicle, we base our approach on an inference engine and a knowledge base, the result of this computation being a file representing the current software architecture of the autonomous vehicle in the syntax of the above tools.

A knowledge base is a notion coming from the domain of Artificial Intelligence: It is composed of a fact base and a rule base, representing objects of discourse (what is known by the system at a given time) and expert inference rules (how knowledge can be inferred upon these objects). The algorithm activating a knowledge base (the inference engine) may use forward (i.e., data driven), backward (i.e., goal driven) or mixed chaining modes.

In our approach, algorithms, e.g., perception ones like lane detection [13], are represented as facts of such a knowledge base: each algorithm of the knowledge base is represented as an object and therefore can be reasoned upon. The environment of the intelligent vehicle (road environment, weather conditions) is also represented as facts in this knowledge base, i.e., objects of classes different than the one of the algorithms. Objects can be dynamically created (inferred) in a knowledge base, enabling to represent different environments of the intelligent vehicle as it follows its path to its destination (e.g., as in the target scenario of section I) and making the inference engine react according to its inference rules.

As opposed to the fact base, the rule base is composed of expert rules encoding expert's knowledge on the assumptions/validity/applicability-zone of each algorithm encoded as fact in the fact base. These inference rules are not supposed to change at run time, except under expert approval. The expertise and knowledge encoded into these rules has been extracted by interviews (with members of our team, designers of algorithms, e.g. [13]), since this is a common practice of knowledge engineers for knowledge acquisition and formalization.

IV. Implementation and Results

The model of section II has been implemented using the knowledge-based system *C Language Integrated Production System* (CLIPS [3]). CLIPS is a forward chaining inference engine with an object-oriented system (i.e., classes and instances), written in the C programming language. CLIPS is connected via XML file, representing a specific software architecture, and libraries/packages, encoding executable code of algorithms, to the proprietary tool RTMaps [11] for activation of the architecture's algorithms and data flow among them, and finally actual control (from sensors to actuators) of an autonomous vehicle.

A. *Inference*

Applicability zones/assumptions of algorithms are encoded as inference rules over a fact base representing algorithms and environmental conditions (i.e., context). An actual inference rule, such as the one for generating the software architecture of Fig. 3, is 1-page long, hence it cannot be shown here due to paper space limitation. Therefore a generic inference rule for generating software architectures as graphs of components and connections (aka nodes and vertices) is presented in Fig. 2

The left-hand side of the generic inference rule of Fig. 2, starting with the CLIPS keyword "defrule" (line 1), is separated from its right-hand side, ending with a closing parenthesis (line 19), by the CLIPS keyword "=>" (line 10).

The left hand side is composed of one environment match on <feature>s and their <status> value (line 2), then pairs algorithm/model matches (resp., lines 3 and 4-5) and finally a finite-state automaton object fsa (line 9) matching the inference state and giving control in the right-hand side to the "printing" state (change of the "status" slot of the object "fsa" in line 18). This pair algorithm/model is repeated as needed by the components/algorithms of the software architecture to be generated --- line 6 for a second algorithm, lines 7-8 for a second model. The name <name1> of the first algorithm (line 3) is the name of first component generated in the right-hand side (line 12). A CLIPS variable ?m1 in the first algorithm (line 3) is used to link the model required by this algorithm to the name of its matched model (line 4) --- similarly for the other pairs algorithm/model in line 6 (variable ?m2) and lines 7-8.

Models preexist in the fact base to store the values of the slots "properties", "types" and "values" required by the data flow design/execution tool RTMaps [11].

Vertices of the graph, i.e., connections between components, are asserted in the right hand side (line 15-17) by linking an output port <oname> (line 16) of an output component <name1> (line 15) to input port <iname> (line 17) of input component <name2> (line 16). The pattern on components (lines 12-14) is repeated as needed for all components of the software architecture to be generated (e.g., 8 components for Fig. 3) --- similarly for connections (line 15-17).

Finally, executable code of each component/algorithm is encapsulated in a package (line 11), and repeated as needed for all components of the software architecture to generate.

```
1 (defrule generate-generic-diagram
2   (environment [(<feature> <status>)]+)
3   (algorithm (name <name1>) (model ?m1))
4   (model (nom ?m1) (properties $?p1s) (types $?t1s)
5     (values $?v1s))
6   [(algorithm (name <name2>) (model ?m2))
7    (model (name ?m2) (properties $?p2s) (types $?t2s)
8     (values $?v2s))]+
9   ?s <- (fsa (status inference))
10  =>
11  [(assert (required-package (name <pck-filename>)))]*
12  [(assert (component  (name <name1>) (modele ?m1)
13      (output-ports <oname>+) (properties ?p1s) (types ?t1s)
14     (values ?v1s)))]+
15  [(assert (connection (component-output <name1>)
16     (output-port <oname>) (input-component <name2>)
17     (start-port <sname>)))]+
18  (modify ?s (status printing))
19  )
```

Fig. 2. Pattern of a CLIPS [1] inference rule to generate a software architecture as a graph composed of components and connexions.

As opposed to Fig. 2, software architectures can also be generated piece by piece: a right-hand side of an inference rule (smaller than the one summed up in Fig. 2) asserts one component only (or a group of components and internal connections), and another rule asserts another component in the same architecture under other environmental conditions. Connections among single/group-of components generated by different rules are asserted inside the most specific of such inference rules.

B. *Results*

If the core knowledge and inference on software architecture generation is described in subsection A, we now describe its test in simulation (replay).

Data logs (i.e., the output ports of some components of an existing software architecture have been recorded) of run of an actual prototype of autonomous vehicle Zoé have been gathered during actual experiments made in Bordeaux, France --- it contains 25575 points. The algorithm for replay is the following:

1. A position in Lambert93 coordinates [8] is extracted from each successive point of the data log. If the current position is too close (i.e., distance less than a threshold) from the previous one, ignore the current point and loop on the next one.
2. The closest point to the one of step 1 is extracted from a topological map of Bordeaux, and its features are extracted (e.g., traffic signs, traffic lights, roundabouts, road curvature, crosswalks, intersections) and entered into the CLIPS fact base as symbolic facts of the knowledge base. If the point in the topological map is the same as the previous one, ignore it (the autonomous vehicle has not moved enough) and loop to step 1.
3. Inference and generation of a candidate software architecture occurs (see subsection A). If the inferred software architecture is the same as the previous one (symbolic test), the context is considered not to have changed and the candidate software architecture is ignored (stability of the software architecture); Loop on the next point in step 1. If no inference rule fires (no software architecture is inferred due to lack of knowledge), the latest software architecture inferred at previous loops, even unadapted, is kept for safety reasons.
4. If the candidate software architecture is considered as new (i.e., the context has changed based on perceived features), generate its XML file (see visualization in e.g. Fig. 3) and run RTMaps in execution mode.
5. Loop on step 1 until all points of the data log are scanned.

The above replay algorithm is implemented as a finite-state automaton using inference rules --- see lines 9 and 18 in Fig. 2 on state inference (corresponding to step 4 in the above algorithm) of the automaton.

As a result, 21 software architectures, including the one in Fig. 3, are successively inferred and activated during the replay of the above data log.

*** faire la manip de Guillaume sur le gros schema RTMaps avec interactions et les 2 Petits. Mesure du temps de réponse.

V. Discussion

- Safety: Safety is a fundamental issue in the domain of Intelligent Transportation Systems. Despite perception's inherent imperfection and potential bugs in embedded source code (eventually due to potentially poor software engineering practices), safety has to be ensured for autonomous vehicles to be accepted by public authorities. To this regard, generating "small" software architectures, as in our approach, leads to easier architecture debugging than with "large" ones. However in our case, safety concerns entail (i) careful analysis and review of algorithms' assumptions to elucidate applicability zones in their input data (this raises difficult questions such as: When does a deep learning perception algorithm fail to recognize e.g. a pedestrian, since machine learning algorithms are trained on a finite (even large) data set?); (ii) careful review of the knowledge base on expert inference rules, to ensure that no lack of knowledge (potentially leading to further expert's interviews and future research directions) can be proved off line --- see the last test of step 3 of the algorithm of section IV.B., aiming at always having an active software architecture on the autonomous vehicle The knowledge represented in knowledge bases on computers cannot be larger than the expert's knowledge it comes from --- one exception is to consider components' parameters values manually determined as a training set and to use supervised learning to estimate unknown parameters value.
- Embedded reasoning: our model, as a reasoning paradigm embedded on an autonomous vehicle, has to be executed on a computer inside the vehicle, e.g., as a software meta-architecture. But since it is based on perception of context/environment for further inferences, it may be integrated as a component in a recursive software architecture that it generates itself, i.e., inference based on context/environment generates a software architecture which includes an inference RTMaps component, located downward perception, which in turn will generate another software architecture including the same inference component, which in turn etc --- a concept similar to *continuation* in functional programming languages (e.g., in the Lisp dialect SCHEME). Or an alternative implementation of our approach could be to encapsulate the CLIPS inference engine and its knowledge base in an RTMaps component, which is connected to the regular components (e.g., SLAM, perception, data fusion, path planning and control), as above, but activates/deactivates possible variant algorithms of each kind with a boolean variable to a specific non-functional input port. This way, a software architecture would be tailorable and tailored at run time by a reasoning component inside itself ---- instead of being fully re-generated each time the context/environment changes, as in our approach.
- Response time: A reasoning algorithm, as an inference engine activating a knowledge base such as in our approach, cannot compete in computational speed and response time with computationally fast perception algorithms, such as [13]. Despite speedups on pattern matching in inference engines, e.g., RETE algorithm [4] and its variants, the response time of a knowledge-based system with an inference engine (including CLIPS' instance) is indeed much slower than fast perception algorithms, leading to fast commands to the autonomous vehicle's actuators --- especially for large knowledge bases. This entails that (i) a software architecture, generated for a context/environment, is always late even if this context/environment is perceived with a small response time; adaptation to context cannot be as fast as perception, it occurs indeed later than perception in the ITS cycle. (ii) A software architecture aiming at embedding reasoning, such as the one envisioned in the previous paragraph, must incorporate algorithms of various response times, the slowest ones configuring the fastest ones --- a knowledge base for configuring parameters' value of algorithms/components.
- Switching vs. evolving: in our approach, a software architecture, once generated, replaces the previous one because of a new perceived context/environment. But, to take an analogy, when a human driver is driving on a highway (initial context), as soon as he is aware of the proximity of an exit to take (e.g., via a traffic sign or a GPS), his perception changes, so as to pay more attention to the right side of the road in his field of view (change of focus of attention), and have no risk to miss the exit, before taking the exit when it arrives (target context). In other words, once a new context/environment is known, even before it is perceived, the software architecture should not discretely switch to the new one, but continuously evolve from the current software architecture towards a new one.

VI. Conclusion

The algorithms (e.g., for perception) activated in an autonomous vehicle in one context (e.g., highway) are not the same as those activated in another one (e.g., urban). Algorithms exhibit applicability zones in their input data, and own assumptions. To make an autonomous vehicle adaptable and context dependent, and thus capable of safely driving in every context/environment encountered towards a destination, we proposed existential software architectures and described a knowledge base of algorithms (e.g., for perception), to infer at run time the software architecture of an autonomous vehicle, considered as a directed graph of nodes (i.e., algorithms) and vertices (i.e., data flow). The dynamic knowledge base has been tested in simulation for replay of data logs of an actual prototype Zoé of autonomous vehicle, which proves the feasibility of the concept.

Future work includes formalizing larger knowledge to infer more algorithms/components, and merging real time performances of successive software architectures using fuzzy logic, as in [9].

Acknowledgment

The authors thank the members of the VEH08 group for numerous fruitful discussions.

References

[1] R. Alami, S. Fleury, M. Ghallab, F. Ingrand. An architecture for autonomy. International Journal of Robotics Research. Vol. 17, no. 4, pp. 315-337, 1998.

[2] Web site of NAVYA projec ARMA. http://navya.tech/?lang=en

[3] CLIPS project web site http://clipsrules.sourceforge.net/

[4] C. Forgy. On the efficient implentation of production systems PhD thesis. C.S. Dept., CMU, 1979.

[5] E. Gat. On three layer architectures. A.I. and Mobile Robotis, D. Kortenkamp. et al. Ed. , pages 195-210, 1998.

[6] B. Hayes-Roth. A blackboard architecture for control. Artificial Intelligence, vol. 26, no. 3, pages 251-321, 1985.

[7] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, M. Balabanovic. A Domain-Specific Software Architecture for Adapative Intelligent Systems. IEEE Trans. on Software Engineering, 4(21):288-301, April 1995.

[8] A. Harmel. Le nouveau système réglementaire Lambert93 (in French). GPS. http://www.geomag.fr/sites/default/files/68_91.pdf

[9] V. Milanes, J. Perez-Rastelli, E. Onieva, C. Gonzalez. Controller for urban intersection based on wireless communication and fuzzy logic. IEEE Trans. on Intelligent Transportation Systems, 2010.

[10] P. Morignot, M. Soury, P. Hède, C. Leroux, H. Vorobieva. Generating scenarios for a mobile robot with an arm. Case study: Assistance for handicapped persones. ICARCV, Singapore, Dec. 2010.

[11] Web site of the proprietary tool *Real-Time Multi-Sensor Advanced Prototyping System* (RTMaps) https://intempora.com/products/rtmaps.html

[12] A. K. Ramaswamy, B. Monsuez, A. Tapus. Solution space modellig for robotic systems. Journal for Software Engineering Robotics (JOSER). 5(1), pp. 89-96, 2014.

[13] M. Revilloud, D. Gruyer, M.-C. Rahal. A new multi agent approach for lane detection and tracking. ICRA, pages 3147-3153, 2016

[14] *Robotic Operating System* (ROS) web site http://www.ros.org/

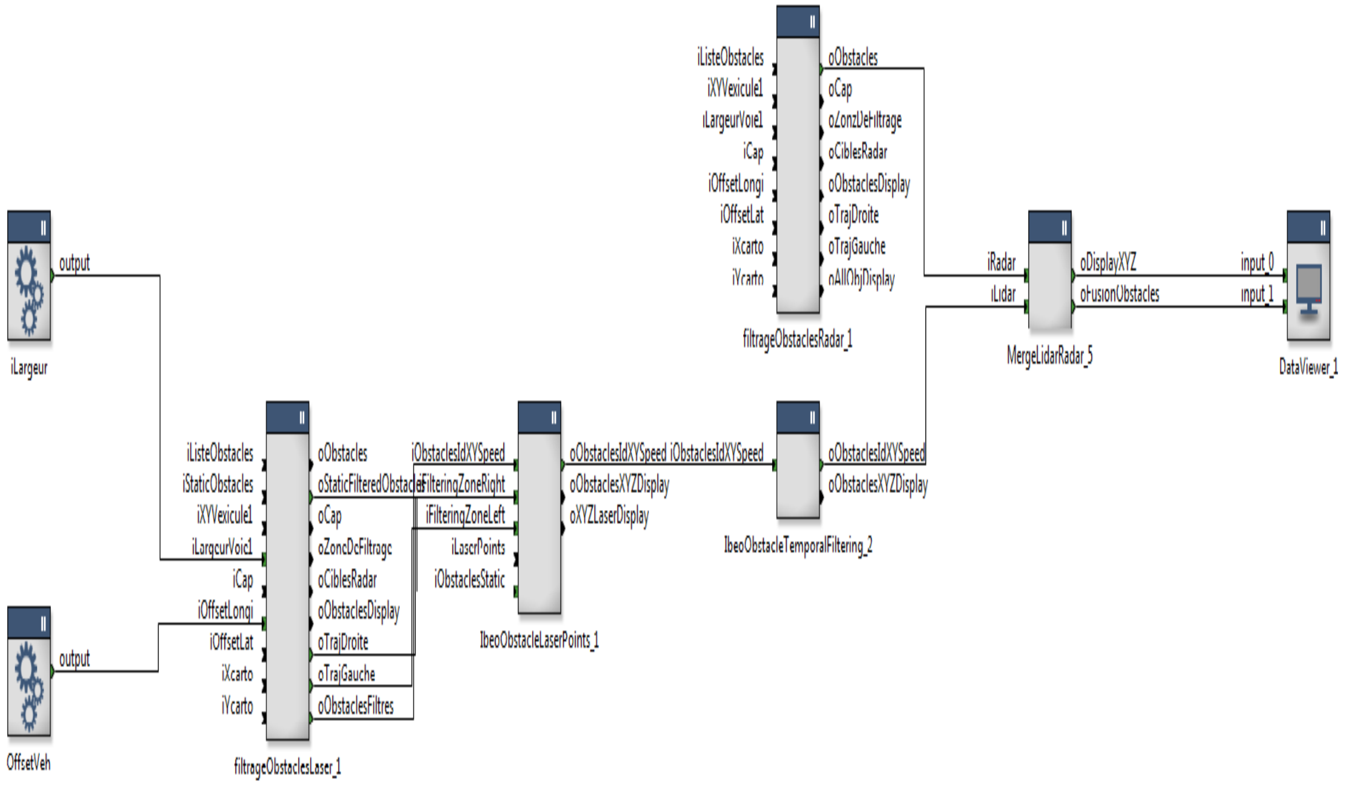

Fig. 3. Visualization of the software architecture for obstacle detection with lidar and radar sensor. 5 inference rules are dedicated to graphical placement of components on RTMaps interface, for debug purposes.