4

Component Based Middleware for Rapid Development of Multi-Modal Applications

Gwenaël Dunand

Intempora, France

4.1 Introduction

Developing multi-modal applications starting from scratch is a tough issue. On the one hand, there are algorithms challenges such as detecting drowsiness or pedestrians in every possible situation. On the other hand, there are programming challenges such as handling multiple sensors data with different frequencies and different nature (video streams, GPS data, laser scans, etc.), as well as implementation details, such as synchronization techniques, multithreading and memory management, for only naming a few.

Moreover, the time required to develop the software is often underestimated [1]. Using an already existing middleware helps to keep on schedule and focus mainly on business problems while decreasing the real-time programming complexity.

There are several middleware that fit all those previous descriptions (ADTF, PolySync, BaseLabs and RTMaps). As RTMaps is the official middleware chosen for the DESERVE project and the author is very familiar with this one, this chapter will sometimes be focused on RTMaps, but other tools might apply as well.

4.2 Using a Middleware

Considering software as layered, middleware incorporates many of these layers vertically. A middleware provides a full, or partial, solution to an area within the application and supplies more than the basic library, it also supplies associated tools like logging, debugging and performance measurement.

Because middleware is vertical system, it may compete or duplicate other parts of the application.

4.3 The Multisensor Problem

The number of sensors used for ADAS applications has increased in the last few years. Now applications use radars, lidars, GPS, high definition stereo cameras, lasers, IMU, CAN Bus, eye trackers, V2V and V2I communication, etc... The problem is how to read all of them within the same application and especially how to synchronize them despite their very different nature (Figure 4.1).

As a matter of fact, most algorithms need to use several sensors to reach a good level of detection. The problem is that those sensors might have different sampling rates, or even worse, event-based outputs. Reading from those sensors simultaneously can be a tricky problem to solve. Let's illustrate this with an example with three signals.

In the Figure 4.2, signal **A** (orange) and signal **B** (green) are periodic with a different period while signal **C** (red) is an event-based signal. One solution would be to use the least common denominator of all sampling rates to perform the reading. While this approach may work with periodic signals like A and B, it won't work with the event-based C signal.

To achieve reading from multi-modal sensors, RTMaps middleware is fully **asynchronous** – each component runs in its own thread – so that any

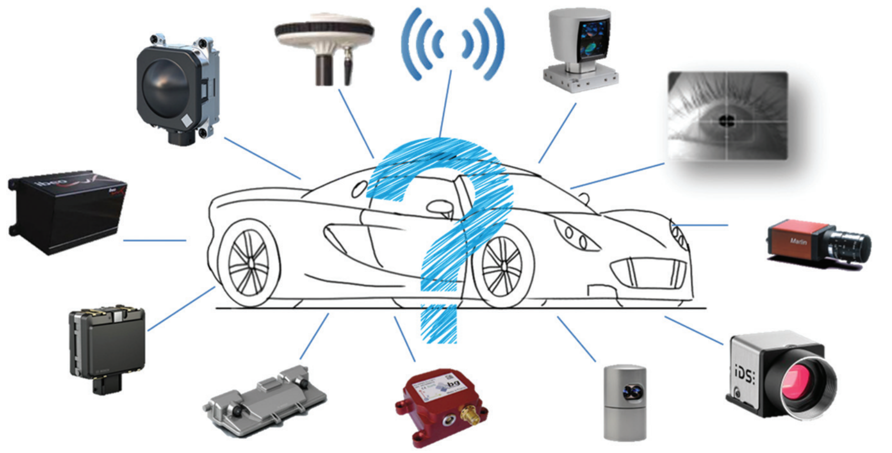


Figure 4.1 ADAS function requires many different type of sensor.

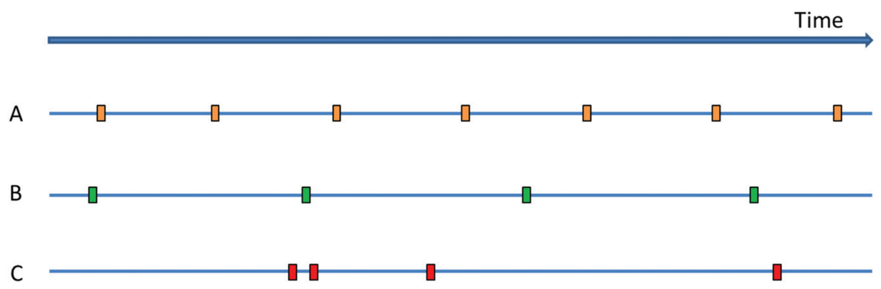


Figure 4.2 Synchronisation issues.

component can react to any data stream, whatever sampling rate it may have. This is the only way to follow the natural pace of each data. This design uses internally blocking calls, removing any extra latency that could happen when using polling methods. RTMaps middleware also defines reading policies to synchronize data streams. While the default policy – *reactive* – works perfectly fine in most case, the user can use one of those:

- *Reactive reading*: a component with multiple inputs will read every time a new data sample is made available on any one of its inputs.
- *Synchronized reading*: a component with multiple inputs will process one sample from each input when data sample with the same timestamps (plus or minus some configurable tolerance) are available on its inputs. This behaviour is made for data fusion and allows re-synchronization of the data streams at any point downstream in the diagram, whatever the latency of the various upstream data channels.
- *Triggered reading*: a component with multiple inputs will read when a new data sample is made available on a given input. It will then resample the data on its other inputs through non-blocking reading.

To sum-up, not only the middleware provides a common platform to build the ADAS application, but it also does take care of the tricky data synchronisation mechanism.

4.3.1 Knowing the Date and Time of Your Data

Using a middleware allows to be very accurate about the timing of your data. For example, RTMaps affects two timestamps to the data: the *timestamp* and the *time of issue*.

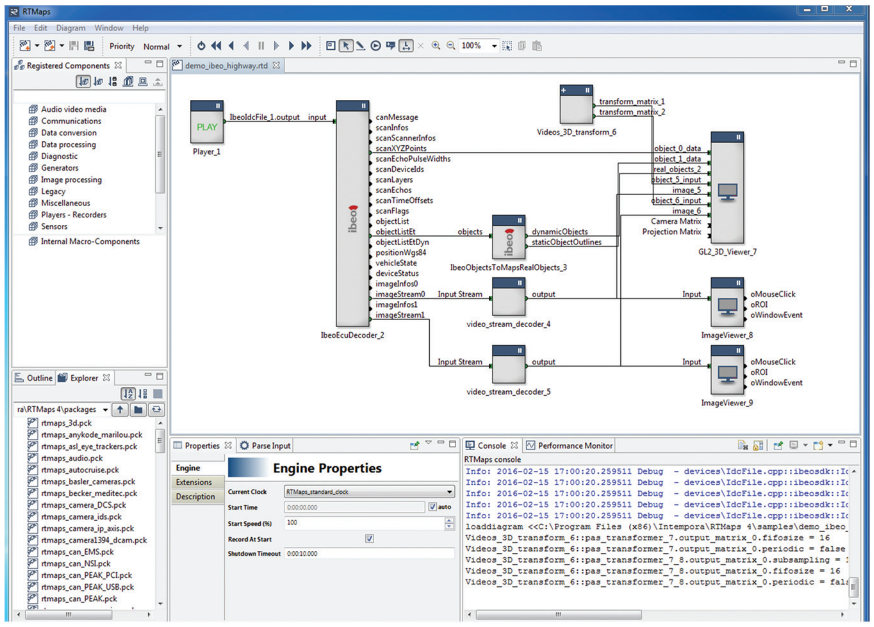
- The *timestamp* is the intrinsic date of the sample. It is as close as possible to the date of occurrence of the real data which the sample corresponds to. It is often supplied by the first component that created the sample (i.e. the acquisition component). The *timestamp* remains unmodified while the sample goes through the different components of the processing chain. The *timestamp* often corresponds to the date where the data is available in system memory.

- The *time of issue* is the date corresponding to the last time the sample was output from a component. Therefore, this date increases as long as the sample runs through the different processing components.

Knowing with precision the time and date of your data is essential to perform synchronized readings (*see previous section*), but it is also useful to estimate the latency of your data or know the processing time of a component which is really vital in real-time applications.

4.3.2 Component-based GUI

RTMaps middleware comes with a user-friendly graphical interface which allows building an application using components (seen as blocks) connected to each other. The Figure 4.3 shows RTMaps studio with a diagram open and a few components in it.



Engine Properties

Current Clock: RTMaps_standard_clock
Start Time: 0:00:00.000 [✓ auto]
Start Speed (%): 100
Record At Start: ✓
Shutdown Timeout: 0:00:10.000

RTMaps console

```
Info: 2016-02-15 17:00:20.259511 Debug  - devices\IdcFile.cpp::ibeosdk::Ic
Info: 2016-02-15 17:00:20.259511 Debug  - devices\IdcFile.cpp::ibeosdk::Ic
Info: 2016-02-15 17:00:20.259511 Debug  - devices\IdcFile.cpp::ibeosdk::Ic
Info: 2016-02-15 17:00:20.259511 Debug  - devices\IdcFile.cpp::ibeosdk::Ic
Info: 2016-02-15 17:00:20.259511 Debug  - devices\IdcFile.cpp::ibeosdk::Ic
loaddiagram <<C:\Program Files (x86)\Intempora\RTMaps 4\samples\demo_ibeo_
Videos_3D_transform_6::pas_transformer_7.output_matrix_0.fifosize = 16
Videos_3D_transform_6::pas_transformer_7.output_matrix_0.periodic = false
Videos_3D_transform_6::pas_transformer_7_8.output_matrix_0.subsampling = 1
Videos_3D_transform_6::pas_transformer_7_8.output_matrix_0.fifosize = 16
Videos_3D_transform_6::pas_transformer_7_8.output_matrix_0.periodic = fals
```

Figure 4.3 The RTMaps Studio.

The advantage of using a graphical user interface is twofold. Firstly, it allows the user to quickly construct an application by using drag and drop techniques and wiring components to each other. Realizing a simple demonstration with a camera and an IMU only takes a few minutes [2] whereas using only hand-written code with dedicated libraries would take weeks.

Secondly, it allows the team to focus on interfaces. This is a very important point since it defines boundaries and clarifies the work between teams. In big projects like the DESERVE project, strict definitions about interfaces are necessary due to the number of partners. The interface for components is composed of inputs, outputs and properties. Once the interface of a task is defined, changing an algorithm for another is not a problem anymore, one component can be replaced by another and the work is done! In the Figure 4.4, the face detection component has one fixed detection interface. The input is YUV image and the output is a vector of rectangle representing the faces found.

Furthermore, the use of macro-components can definitely simplify the diagram by splitting the global problem into sub-problems (Figure 4.4). All the implementation is hidden in first appearance to simplify the reading, but of course looking under the mask would reveal all the internal details.

4.3.3 The Off-the-Shelf Component Library

The off-the-self component library represents all the already available components in the middleware. This is an important part of it because it allows accelerating the application development by using and reusing already developed component. Here are a few categories of components:

- *Sensor interface*: This category represents all the components that allow to read/write from/to a sensor. When a sensor is present in the library, the user has just to drop a corresponding component on the current diagram and configure it to retrieve the data. That work can be done easily with a consequent time benefit.

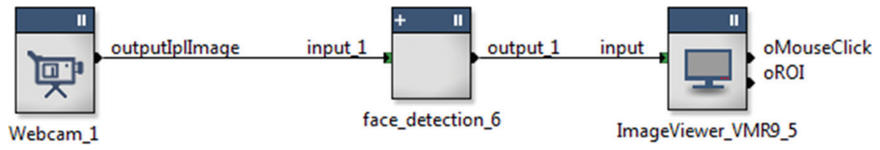


Figure 4.4 Components and interfaces.

- *Data generators*: When comes the time of testing a component, it might be useful to emulate a missing sensor with random generated data (vectors, CAN frames, images). This does not replace real sensors but it can be enough sometimes.
- *Viewers*: Very important libraries, which allow displaying information about data stream during the execution (images, vectors, CAN frames…). As an example, the *DataViewer* (Figure 4.5) can display generic information (timestamps, size, etc.) and specific ones (width and height of an image if current data is an image) as a tree. This is very useful to inspect data along a processing chain and check that such component behaves correctly.
- *Player and Recorder*: Those components allow to record and replay any data stream. Using a recorder, the user is able to record any scenario (outdoor session, motorway driving test, automatic car parking, etc.) and replay it at the office with the exact same data and timestamps.

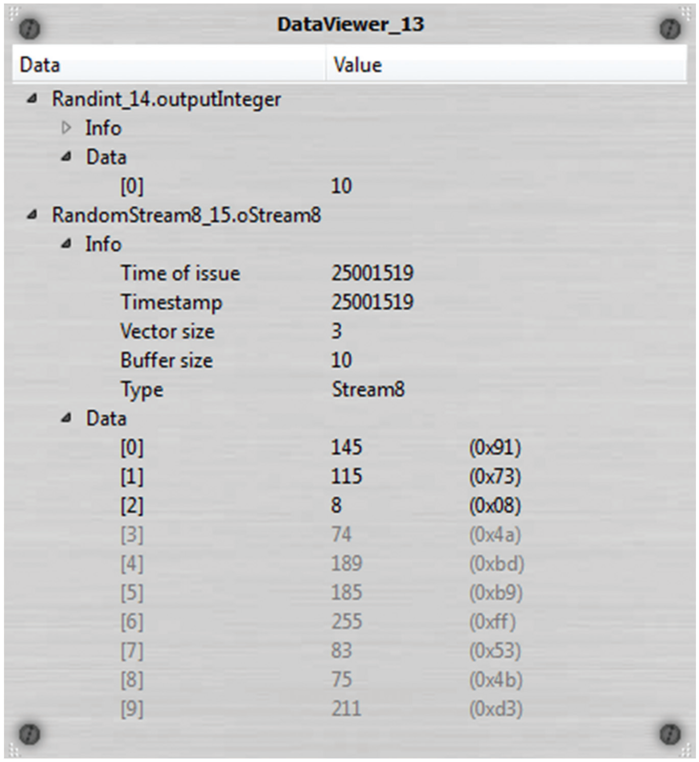

```
DataViewer_13
Data                          Value
◢ Randint_14.outputInteger
   ▷ Info
   ◢ Data
        [0]                   10
◢ RandomStream8_15.oStream8
   ◢ Info
        Time of issue         25001519
        Timestamp             25001519
        Vector size           3
        Buffer size           10
        Type                  Stream8
   ◢ Data
        [0]                   145        (0x91)
        [1]                   115        (0x73)
        [2]                   8          (0x08)
        [3]                   74         (0x4a)
        [4]                   189        (0xbd)
        [5]                   185        (0xb9)
        [6]                   255        (0xff)
        [7]                   83         (0x53)
        [8]                   75         (0x4b)
        [9]                   211        (0xd3)
```

Figure 4.5 Inspecting data with the data viewer.

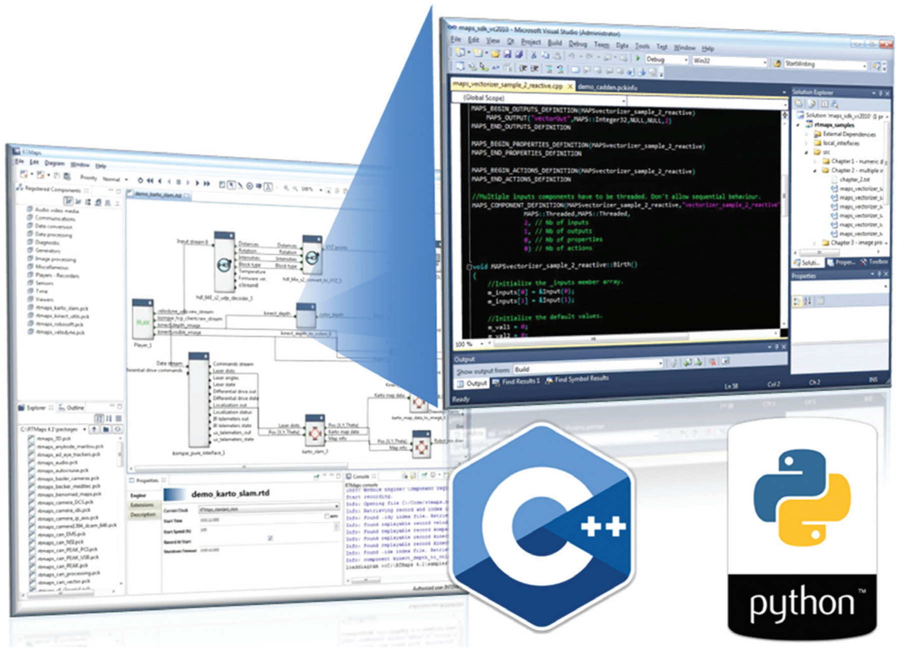
4.3.4 Custom Extensions

Extending the component library is done through the **SDK**, whose purpose is to expand the capabilities of the middleware by the creation of new components. In RTMaps for example, the SDK is available for both C++ and Python (Figure 4.6). Thanks to this SDK, the user can integrate his own code into a component and use it directly in this diagram.

Once a new component has been created, it can be shared with others. When using C++, each component is compiled code which means that only the binary code is used in the middleware and so the IP is preserved. Anybody can share his work while keeping the source secret.

4.3.5 About Performance

Using a high performance middleware is still essential nowadays. Indeed, even if the power of the computer tends to increase continuously, the trend is to run applications on embedded systems with the smallest footprint possible. The explanation of this trend is quite simple: the prototype vehicle has to be as close as possible as the real vehicle. In many companies, no desktop computer


maps_sdk_vc2010 - Microsoft Visual Studio (Administrator)
maps_vectorizer_sample_2_reactive.cpp
demo_cadden.pckinfo
(Global Scope)
demo_karto_slam.rtd
python

```
MAPS_BEGIN_OUTPUTS_DEFINITION(MAPSvectorizer_sample_2_reactive)
    MAPS_OUTPUT("vectorOut",MAPS::Integer32,NULL,NULL,2)
MAPS_END_OUTPUTS_DEFINITION

MAPS_BEGIN_PROPERTIES_DEFINITION(MAPSvectorizer_sample_2_reactive)
MAPS_END_PROPERTIES_DEFINITION

MAPS_BEGIN_ACTIONS_DEFINITION(MAPSvectorizer_sample_2_reactive)
MAPS_END_ACTIONS_DEFINITION

//Multiple inputs components have to be threaded. Don't allow sequential behaviour.
MAPS_COMPONENT_DEFINITION(MAPSvectorizer_sample_2_reactive,"vectorizer_sample_2_reactive"
            MAPS::Threaded,MAPS::Threaded,
            2, // Nb of inputs
            1, // Nb of outputs
            0, // Nb of properties
            0) // Nb of actions

void MAPSvectorizer_sample_2_reactive::Birth()
{
    //Initialize the _inputs member array.
    m_inputs[0] = &Input(0);
    m_inputs[1] = &Input(1);

    //Initialize the default values.
    m_val1 = 0;
```

Figure 4.6 Developing a new component.

in the trunk of the car are allowed anymore, all systems have to be (or at least look) embedded.

Furthermore, the middleware is pushed further and further in the development chain. A few years ago, most of the middleware were assigned to do only prototyping and once the prototype application was finished, all the work had to be done again on dedicated hardware. This not the case anymore, now the middleware should be able to run on low consumption cards that equip pre-series cars.

Consequently, OEMs are looking for high performance middleware that runs on small form factor cards as well as on Personal Computer so that working on lab or real scenarios makes no difference.

4.4 Compatibility with Other Tools

4.4.1 dSPACE Prototyping Systems

In the frame of the DESERVE project, a bridge has been developed between the dSPACE MicroAutoBox and RTMaps (Figure 4.7). The dSPACE MicroAutobox is the de facto standard for real-time control loop such as chassis control, body control and powertrain. Combining this dSPACE prototyping system to the RTMaps middleware provides an extremely powerful framework capable of doing multisensor acquisition, data processing and controlling actuators in a hard real-time way.

The MicroAutoBox typically serves as an embedded controller to process the ADAS application algorithms in real-time and to interface the vehicle bus, sensors and actuators. It is a prototyping ECU with a predefined set of I/O which is qualified for in-vehicle use.

In the context of the DESERVE project this platform was extended by an Embedded PC and an FPGA Board. The embedded PC features a multi-core Intel® Core^{TM} i7 processor running at 2.5/3.2 GHz and the connection to the actual embedded controller is implemented via an internal Gigabit Ethernet

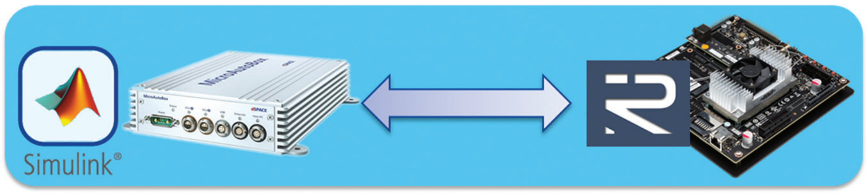


Figure 4.7 dSPACE MicroAutobox and RTMaps Bridge.

interface. The embedded PC integrated in the MicroAutoBox can be used to flexibly run any x86 based development framework available for prototyping perception and fusion algorithms, such as RTMaps, and to exchange easily data with the embedded controller [3].

4.4.2 Simulators

ADAS are becoming more and more promoted because several key functions permit to increase the level of vehicle safety. Most of the time, it is a challenge to access to the equipment and sensors information on vehicles, making difficult to design and test these new algorithms. Some of the applications are based on perception sensors embarked on the vehicle, which interact with the vehicle, driver and environment through electronic control units. For those reasons, the simulations of the algorithms and the analysis of existing solutions for virtual testing are very important tasks.

Using simulators has many advantages: tune the scenario at will (add rain or fog like in Figure 4.8), test dangerous situations where real data is hard to get, use the output of any algorithm to modify the scenario of the simulator (close the loop), etc. It's pretty much a fact now; virtual testing allows massive

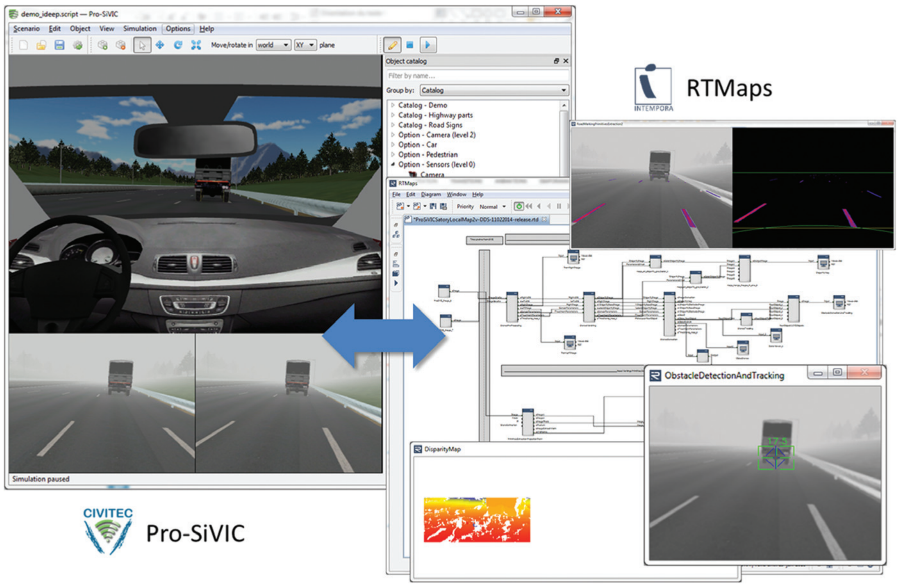


Figure 4.8 ProSivic working together with RTMaps.

reduction cost. In the DESERVE project, many simulators have been used in collaboration with RTMaps: ProSivic [4], dSPACE ASM [5], etc.

4.4.3 Other Standards

Middleware supports other standards as well. RTMaps implements the DDS [6] standard interface via the Prismtech OpenSpliceDDS implementation. This is very convenient to stream data from RTMaps to anywhere and vice-versa. This DDS interface was developed in the frame of the DESERVE project.

Other standard protocols are also supported, like XIL or XCP, which allow manipulating RTMaps with off-the-self tools that implements those protocols themselves.

Of course, most of the middleware on the market will also support NMEA, CAN/DBC, RTSP, I2C, GPS, SIP, TCP and UDP as well. The compatibility with major industry standards is essential so that the middleware interacts painlessly with other tools.

4.5 Conclusion

Most DESERVE partners have been using RTMaps and ADTF middleware as the common perception platform to speed up their development processes and exchange components between each other.

Indeed, partners like Continental, FICOSA, Vislab and CTAG have encapsulated their acquisition routines and custom algorithms into RTMaps components, which in turn have been integrated into a global acquisition and processing diagram by other partners (OEMs most of the time). This modular approach made the collaboration easier between a large number of partners, which was one of the difficulties of the DESERVE project.

Another example, CRF (*Centro Ricerche Fiat*) has used RTMaps and the bridge to the MicroAutoBox – developed in the frame of the DESERVE project – for their emergency breaking application. The sensor acquisition, the pedestrian detection, information display and the breaking order are done *via* RTMaps.

As a conclusion, in the DESERVE project, having a middleware has allowed engineers to focus on their main activity – obviously ADAS functions here – and not on advanced programming issues, but it was also very helpful to exchange components between partners.

References

[1] Software Engineering 8th Edition, p. 109, ISBN-13: 978-0321313799, 2006.

[2] Intempora. (2012, February 20). *RTMaps4 demo* [Video File]. Retrieved from https://www.youtube.com/watch?v=HBxFq04S91g

[3] Joshué Pérez Rastelli, David Gonzalez Bautista, Fawzi Nashashibi, Fabio Tango, Nereo Pallaro, et al. Development and Design of a Platform for Arbitration and Sharing Control Applications – a DESERVE approach-. IEEE SAMOS Conference, Jul 2014, Samos, Greece, pp. 322–328.

[4] Prosivic. (2016, June 21). Retrieved from http://www.civitec.com/

[5] dSPACE. (2016, July 12). *Simulation tool suite*. Retrieved from https://www.dspace.com/en/inc/home/products/sw/automotive_simulation_models.cfm

[6] OMG. (2016, July 11). *DDS: the proven data connectivity standard for the IoT*. Retrieved from http://portals.omg.org/dds/