# Dynamic software updates vs AUTOSAR

Hélène Martorell[1,2,3], Jean-Charles Fabre[2,3], Matthieu Roy[2,4], and Régis Valentin[1]

[1] RENAULT Technocentre, 1, Avenue du Golf 78288 Guyancourt, France
{helene.martorell, regis.valentin}@renault.com,
[2] LAAS-CNRS, 7 avenue du colonel Roche, Toulouse, France
{martorell, fabre, roy}@laas.fr
[3] Univ. de Toulouse, INP, LAAS, F-31400 Toulouse, France
[4] Univ. de Toulouse, LAAS, F-31400 Toulouse, France

**Abstract.** Automotive systems are now shifting to become more software-based machines. The AUTOSAR (AUTomotive Open System ARchitecture) standard defines a software architecture, which remains quite static: this is one of its main drawbacks. Before a vehicle is put in operation, the whole software system is tested, validated and then uploaded in ECUs in a monolithic fashion. Improving adaptability of software system would allow car manufacturers to easily modify the initial configuration or provide additional features to the customers. For this reason after a careful analysis of the main concepts of the standard, we defined an approach for allowing dynamic updates. Currently such modifications are not supported by the standard, but could lead to significant gain in time and resources. Based on this approach we analysed the undesired events that could arise in consequence of the mechanisms added for the updates. Then safety features are suggested in order to help preventing these hazards, in terms of prevention and tolerance.

Keywords - Dynamic Update, AUTOSAR, Automotive, Embedded Systems, Safety

## 1 Introduction

Todays vehicles are becoming increasingly software intensive. In this field, quality, cost and time-to-market criterion are paramount. For this reason the AUTOSAR [1] [2] (AUTomotive Open System Architecture) standard was designed. It defines a component-based software architecture that aims at increasing reuse by abstracting the hardware away.

AUTOSAR presents a static architecture since everything must be known at compile time, before the ECUs (Electronic Control Units) are loaded with the software. This means that adding new functionalities is not straightforward. Yet, it is becoming necessary : it will for example allow the car's owner to benefit from features that did not exist when his car was produced. A consequence of this kind of technique could be a reduction of the amount of software uploaded on production line and a specialization of the ECU in order to fulfil clients customization wishes [3]. A simple example in which this approach could be used is adding a pulse electric window control in order to increase comfort and safety.

Moreover this will guarantee the car owner to have the most recent version of all functionalities in his vehicle at any given time whereas charging the whole application at once could result in having some obsolete ones.

The contribution of this paper consists in the presentation of an approach for allowing partial and dynamic updates a posteriori, i.e. to prevent from reloading the whole application software for a limited update in an AUTOSAR platform. This allows for incremental updates: only the required functionalities are uploaded or upgraded.

Note that the objective is not only to allow for adding new function but also to update existing one. The mechanisms used for both approach are identical.

It relies on a number of concepts and tool. Indeed, AUTOSAR requires a specific development process to be followed and our approach will result in minor changes in this process. The latter are handled with specific tools that we developed. The key concept for allowing updates is the introduction of specific placeholders when designing original application, called containers, which can later be filled in with new functionalities.

To complement these mechanisms for updates, specific safety features are also desirable. Usually a number of specific mechanisms for safety are introduced when creating the automotive system. Nevertheless, since we aim at introducing new concepts and eventually execute new code in the system, it is also important to sum up the available features and add new ones if necessary. Indeed, the final application should fulfil the requirements and methods of ISO 26262.

The paper is organized as follows: Section 2 presents briefly the key concepts, abstracted from the AUTOSAR standard. We then present the overall approach itself in Section 3, describe shortly our design for adaptation in Section 3.3 and apply this approach to a case study in Section 4. Finally, address related works in Section 6 and conclude.

## 2 Context

This section presents the different concepts required to define our methodology. It is based on the AUTOSAR standard, that defines *i*) the different concepts of automotive embedded software architectures and *ii*) AUTOSAR OS, mainly based on OSEK [4], which was the previous standard defining the characteristics of OS used in vehicles.

The presentation of AUTOSAR and AUTOSAR OS in this section is oriented towards the identification of the key elements for defining adaptation areas. It is certainly brief and incomplete but it only aims at identifying the different dimensions of an application which will be necessary for modelling AUTOSAR based applications and defining a reference for adaptation.

### 2.1 Architecture

AUTOSAR presents a layered architecture that allows abstracting away from the hardware. It consists in 4 different levels. The bottom one is the hardware, on top of which stands the basic software. The latter contains low-level drivers and services along with the operating system. Then there is the RTE (Run-Time Environment), which acts as an ad-hoc middleware that handles communications in the ECU (Electronic Control Unit). Finally the top layer contains the Software Components (SWC) that use the services offered by the lower layers. The SWC corresponds to the functions realized by the ECU often called the “applicative software”.

### 2.2 Structural Concepts

**RTE (Run-Time Environment)** It corresponds to glue code specifically generated for each ECU in order to handle communications. Communication can either be between different SWCs or between SWCs and the Basic Software.

**SWC (Software Component)** SWC do not have any existence in the final implementation: they are made of *runnables*: piece of software that actually realizes functions. The runnables are actually mapped onto the tasks of the OS for execution.

**Communication** There are two types of communication: *Implicit* for which data are read at the beginning and written at the end of an execution instance and *Explicit* for which data are read when needed and written when produced during the execution. All data are passed through the RTE.

### 2.3 Runnables

Based on their communication requirements, the runnables can be divided into 3 categories :

- Cat. 1A: *Implicit* Data.
- Cat. 1B: *Explicit* Data.
- Cat. 2: *Explicit* Data and can use extra synchronization point in its execution (to wait for an external event).

This distinction corresponds to the different timing behaviour. When using implicit data, the runnable will complete in a finite amount of time. Explicit data, on the other hand will result in calling an external function for which response time is not known by its caller. Finally when the runnable can wait for an external event, it is hard to predict when this event will arise (e.g. if the expected event is the user pressing the warn lights button).

### 2.4 Tasks

To be executed runnables need to be allocated into an OS task. There are two kinds of tasks basic and extended. Basic tasks can only be in 3 states : *running*, *suspended* or *ready*. Therefore they cannot wait for external events. Extended tasks have one extra state : *waiting* which allows them to wait for an external event before resuming their execution.

The category of runnable will determine the kind of task the runnable can be mapped onto: Cat. 2 runnables can only be mapped onto extended tasks.

## 3 Overall Approach

In this work the updates only focus on the application layer. As explained in section 2.2 components in AUTOSAR are just a collection of runnable. Therefore we aim here at updating one runnable to begin with. This way, updating several runnables could amount to updating a complete SWC. We focus here on an incremental update: adding new functionalities or upgrading existing one (for example, for a more up-to-date version) in an embedded application running in an ECU. Nevertheless both these changes use the same update mechanisms and therefore upgrade are handled in the same way as updates. Therefore they will not be differentiated in the rest of this paper.

### 3.1 Adaptation Area

Partial dynamic updates are not supported in AUTOSAR specifications: everything must be defined at design time for generating the RTE. Therefore we have to integrate in the process the mechanisms for future updates.

This is the reason why we introduced a new concept called *Adaptation Areas*. They correspond to a set of specific properties that can define a space in the application. The concepts defined in section 2 help us design these areas. We will focus here on an adaptation area for one runnable since this is the desired granularity for updates.

Characteristics for the adaptation areas are created on a structural level. The relevant features for update are the following:

- Activation mode and Trigger: is the runnable periodic or sporadic and what kind of event will trigger its execution?
- Category: this depends on the characteristics of the runnable : the kind of communication it uses (implicit or explicit, see 2.3) and the presence or absence of wait points.
- Data and Access mode: the runnable needs to communicate with its environment, this corresponds to the data used for communication and their mode (implicit or explicit).

Fig.1a shows these various characteristics on a radar. Taking a value on each axis would allow to define an adaptation area from a structural standpoint.

### 3.2 Container and parameters

We call *container* the implementation counterpart of the adaptation area. That is to say it is a physical representation placed in the application with specific characteristics. Its role is to be later filled with updates.

Containers are introduced in the application at design time and represent an empty space for future updates. A specific container will corresponds to all the structural features defined by its adaptation area. In addition to these features it will have to be integrated at implementation level (*implementation parameters*). That is to say containers have to be scheduled for future execution, and time must be planned ahead for them. A container is therefore located into a task and is executed at run-time. It is initially empty.

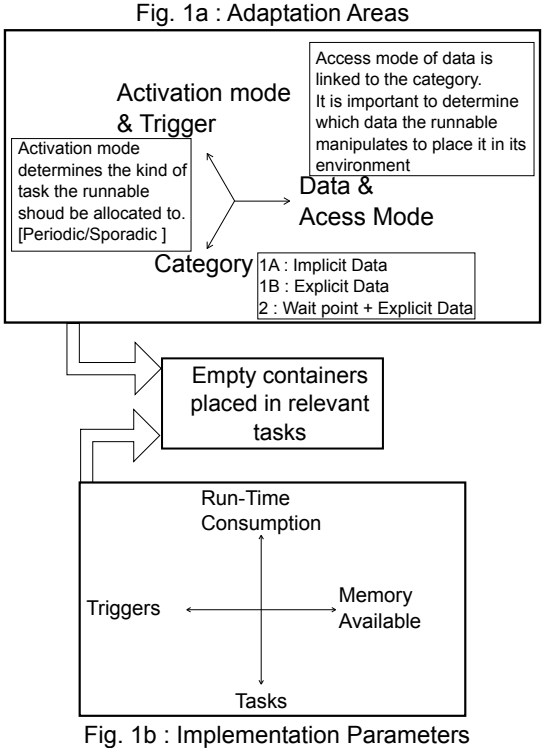


**Fig. 1.** Container Definition

Based on our study of the standard, we determined two level of design that have direct impact on

the containers : structural and implementation. Firstly containers aim at harbouring new runnables. Therefore they will represent placeholder for a runnable with corresponding structural characteristics. There characteristics are those of the adaptation areas described in section 3.1 (activation mode, data and access mode, activation mode and trigger).

Implementation characteristics also have an influence on the location of the empty container. In particular, for the subsequent update memory consumption and Worst Case Execution Time (WCET) need to be determined. This is necessary for integrating the update in an existing and running application. Fig. 1b shows the specific implementation characteristics for an automotive application. Containers must fit in within these characteristics for being fully integrated. That is to say, containers must have a defined run-time consumption (we use here their WCET), and they must belong to a specific task with all its characteristics (period (if periodic) or event that activate it (if sporadic), priority, preemptive or not) and that will activated by a trigger. The memory space then corresponds to the available memory for containers after the application is loaded in the ECU. Note that when adding an update inside a container run-time characteristics have to be taken into account : that is to say we must know the desired schedule for the update so that it will execute at the appropriate time regarding its surrounding runnables and the required communications.

Finally, based on both the structural and the implementation characteristics, the containers are comprehensively defined and can be integrated in the application and loaded within it in the ECU. Therefore the base application will execute exactly as if containers were not there.

As already mentioned, in this work we aim not only at allowing updates, i.e. the addition of new functionalities in the ECU, but also upgrade. The latter corresponds to the replacement of an existing functionality by a more up-to date version. Both these possibilities rely on the same underlying implementation.

### 3.3 Design for adaptation

In this work several hypothesis are made: firstly we are using a pre-wired approach. This means that every degree of freedom is added at design time. Therefore when the update happens, verification of availability should be performed, but the mechanisms are already integrated in the application. Besides, RTE and Basic Software are considered as fixed as we only aimed at adding new software functionalities. This means that it is not possible to add new communications channels within the RTE or to get communications from other ECUs that are not intended to our ECU. Therefore, since an update will need to communicate with its environment, it will reuse exising channels. This can lead to problems that will be described subsequently.

AUTOSAR defines specific processes for creating automotive embedded software that complies with the standard. For this reason, the changes done to the process have to be automated in order to fit in the tool chain [5]. Fig. 2 shows a simplified development process for automotive embedded software. It shows that, based on specifications, a global model for the software is derived. Then from this model an AUTOSAR model is designed. Note that these two model usually represents software distributed on several ECUs. Then for each ECU

We can see here that there are two steps that need modifying. The first one is upstream and consists in adding the containers. This way since the next steps will rely on informations of the *Software Functional Model* the containers will also exists in subsequent steps. Then an *AUTOSAR model*, from which the lower layers of the architecture are generated, needs to be developed. In parallel of this, a *functional model* is created for obtaining the *Applicative Software*. The second level of changes occurs after the generation of lower level software. Indeed, implementation changes have to be made for adding the actual low-level update mechanisms.

In order to comply with the tool-based AUTOSAR process, we defined new ones for performing the previously described changes in the process. The first one will allow to add the containers in the

application. Then there is a set of tools for adding the specific mechanisms automatically. This set of tools needs an access to the RTE source code.

It is worth noting that these tools can be used on different automotive application that comply with the AUTOSAR standard.

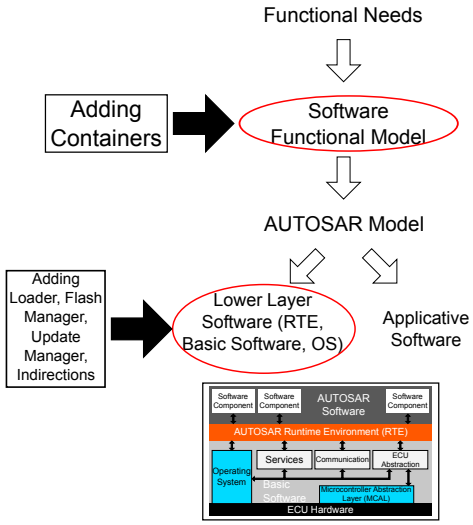


**Fig. 2.** Simplified Development process for an automotive application

One of the main ideas for allowing effective changes at run-time is the addition of an extra level of indirection between the runnable and its caller. This will enable us to change the runnable without modifying the general structure. This is also the technique used for filling in the containers. This level of indirection is added off-line when preparing the application.

During the lifetime of the system, after the ECU is uploaded with the base application and the container, update can then take place. When an update occurs, a container with characteristics matching those of the update runnable has to be found by the update manager for uploading. *Update Manager* actually groups several functions. The first one is the *loader* that enables to upload the update runnable into the ECU from an external source. The second one is the *Flash Manager* that will copy the runnable in the appropriate location of the ECU. Finally the actual update manager handles the on-line checks before allowing the update (e.g. finding a suitable container, checking the memory space, etc) and the after the upload is performed (checking that the update was properly integrated in the application).

## 4 Case Study

In order to test our approach we extracted from an ECU similar to BCM (Body Control Module) a simple application that we modified. This application is used for controlling the blinkers in a car. It reads from the sensors (turn switch sensors and warn light sensor), proceeds the received data and sends a signal to the actuators in order to trigger the flashing of the light bulbs. In our case, for demonstration purposes we used the buttons and LEDs provided by the extension board of the microcontroller. The test target was a PowerPc type 5510.

We created containers placed into tasks that could later be used for updates. We identified two main case study that we deemed interesting regarding the blinkers : adding impulse control blinker and emergency brake warning. The first case can be described as follow : when a short impulse is detected on the left or right turn switch sensor, then the blinker should blink 3 times. This can be used for example when changing lane. The second case is more related to safety : when the driver brakes suddenly, the warning should be trigger until the car restarts.

This two case study enable to experiment on the both approach : the first case will result in upgrading an existing functionality in order to improve it and the second one will add new functionality using a container.

Run-time impact of the update mechanisms is very limited : the first one is the dereference of function pointers which is negligible, and the second one is the execution of the update manager which is limited since it seldom executes. The update themselves have an impact on the runtime, but this is taken into account at design time since specific time slots are left available for this purpose.

Memory-wise the update will be stored in available memory, therefore limiting their impact. Memory consumption is also increased by the meta-data necessary for each container and existing runnables, and by the size of the update manager. The latter is fixed and therefore when the application grows its impact will be increasingly negligible. The meta data represent a very limited amount of memory, but will grow linearly with the application (more runnable and more containers will mean more meta-data).

## 5 Safety mechanisms

The added update mechanisms and the actual updates that are inserted afterwards in the application could lead to undesired events. Thus, their consequences should be studied beforehand and features added for guaranteeing that the updates will not harm the system.

There are two levels of tests for the updates. The first level is off-line for making sure that it fulfils the correct properties and that its functional behaviour is consistent. The second one is on-line for verifying the update in its context and making sure that the update runnable can be fit in the application.

It is worth noting that the new version of the system, including the update must be validated by the manufacturer testing process off-line. However, some runtime problems may affect these additional application features. It is thus reasonable to encapsulate the updates by means of fault containment wrappers.

Tests should be first performed at upload time: the system should check its ability to receive the new runnable. Not only memory-wise, but also from a structural and a run-time point of view. The memory criterion is the simplest one: is there enough available space in the ECU for loading the update. Then the structural one corresponds to the presence of an empty container that has matching characteristics. Finally on a run-time point of view, we must make sure that the WCET of this update can be fit in the schedule.

One of the important mechanism that should be studied is the reuse of existing communication mechanisms. Channel can be used either to read data or to write them. For reading purposes, if two runnables have to read the same data in order to make a decision, this data should not only be consistent for both of them, but also available. That is to say the reading of the data should not be destructive and the synchronization should be appropriate. When it comes to writing, the main concern is data racing : if two runnables are using the same channel, arbitration is necessary to determine which should be using the channel. This arbitration can for example be implemented using a third-party as an arbitrator.

## 6 Related Work

The field of component-based architectures and dynamic updates of software has been researched for a long time: McIllroy was the first to describe it in 1969 [6] and ever since component-oriented software have been extensively developed in various fields. Indeed, it provides with more adaptable software that will enable cheaper maintenance, better ability to cope with complexity, to increase the quality and evolution of the software [7].

Regarding methods for dynamic updates based on components, there are three major types of approaches: routine-based update, component-based update and updates at the granularity the whole program.

Routine-based update corresponds to a finer granularity as it typically updates individual functions or objects. For example Ksplice [8] uses a system of patches for hot updates on operating system's kernels without reboot, and replaces entire functions. Our approach does not focus on the operating system but instead on the applicative and middleware level. Ginseng [9] explores the same concepts: using patches for dynamic updates of C programs, in order to perform fine-grained updates while insuring a continuity for the state of the program. Yet, this approach requires access to the source code of the application and AUTOSAR allows both source code and object code with appropriate XML description for SWC. Besides, this approach was not designed for embedded systems either. Nevertheless the underlying concept that we want to explore for allowing dynamic update in an automotive embedded context are similar. That is to say we need to make the code dynamically updatable. It is worth noting that the AUTOSAR methodology is built around a tool-chain [5], which means that dynamic update will require to add an extra step.

In [10], Li *et al.* present an OSGI-based automotive specification. OSGI is a component-based platform that enables to download, install and uninstall bundle(service application). However their approach does not comply with the AUTOSAR specification and it relies on the Java language. Besides it focuses on the infotainment part of the automotive system. Another component-based approach for embedded systems is described in [11]. However in their approach they introduce a component manager that is itself an updatable component, to handle dynamic update and wiring for the other component. They present update algorithm, state transfer and specific update points in the execution of the program.

On the other hand, in [12], several methods for dynamic updates in component-oriented embedded systems are presented. Yet, none of them is specifically designed for automotive embedded systems.

In [11], the authors present a framework that enables dynamic update for component-based embedded system. However in their approach they introduce a component manager that is itself an updatable component, to handle dynamic update and wiring for the other component. They present update algorithm, state transfer and specific update points in the execution of the program.

Safety-wise various approaches exist when it comes to adding safety mechanisms either in automotive [13] [14] or more generally for component-based system [15]. However, we need here an ad-hoc approach that can reuse some of these mechanisms but for in a targeted way for guaranteeing the safety of the added updates.

## 7 Conclusion

This paper presents a brief overview the AUTOSAR-related concepts relevant for allowing dynamic update of software in an AUTOSAR application. We also presented the model we built for designing *adaptation areas* and the associated *containers*, which are implementation counterparts of adaptation areas. These container correspond to placeholders in the application for future updates. Details of the design for adaptation and applying the concepts on a specific example were also treated in the last sections of this paper. In this work, we use a "pre-wired" approach, that is to say we define a priori,

when conceiving the application, some adaptation containers that can afterwards be filled in with new runnables.

Then when all the dimensions for the desired container are known, it can be integrated in the actual embedded application and latter used for storing and executing update runnables. It is relevant to place several container presenting different characteristics in the application to perform several updates.

In this paper, we focused on the concepts that are necessary for dynamic update and a model for empty containers that can be placed into wisely chosen location for allowing posterior dynamic updates. The adaptation engine that will perform the actual update is beyond the scope of this paper. Nevertheless, this engine should handle several contingencies such as keeping track of the available containers, detecting new updates available and loading them or modifying the application for executing the updates.

It is also important to highlight that dynamic updates have to be carefully monitored for safety purposes: the update should not prevent the system from working properly. We present here briefly the contingencies that are related to the use of containers for the updates.

## References


1. AUTOSAR Development Cooperation, http://www.autosar.org.
2. S. Furst, J. Mossinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkmper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar - a worldwide standard is on the road," International VDI Congress Electronic Systems for Vehicles, 2009.
3. S. Mollman, "From cars to tvs, apps are spreading to the real world," *CNN*, October 2009, http://edition.cnn.com/2009/TECH/10/08/apps.realworld/.
4. OSEK Group, "Osek/vdx operating system (release 2.2.3)," 2005, http://portal.osek-vdx.org/.
5. S. Voget and P. Favrais, "Autosar and the automotive tool chain," *Methodology*, 2010.
6. M. D. McIlroy, "Mass-produced software components," *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.
7. I. Crnkovic, "Component-based software engineering - new challenges in software development," in *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, june 2003, pp. 9 – 18.
8. J. Arnold and M. F. Kaashoek, "Ksplice: automatic rebootless kernel updates," in *In Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 187 – 198.
9. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006, pp. 72–83.
10. Y. Li, F. Wang, F. He, and Z. Li, "Osgi-based service gateway architecture for intelligent automobiles," in *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, june 2005, pp. 861 – 865.
11. M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '09. New York, NY, USA: ACM, 2009, pp. 2:1–2:6.
12. B. Y. Vandewoude Yves, "An overview and assessment of dynamic update methods for component-oriented embedded systems,," in *proceedings of The international Conference on Software Engineering Research and Practice*, Las Vegas USA, 2002, pp. 521–527.
13. C. Lu, J.-C. Fabre, and M.-O. Killijian, "An approach for improving Fault-Tolerance in Automotive Modular Embedded Software," in *17th International Conference on Real-Time and Network Systems*, Paris, France, 2009, pp. 132–147.
14. T. Piper, S. Winter, P. Manns, and N. Suri, "Instrumenting autosar for dependability assessment: A guidance framework," in *DSN*, 2012, pp. 1–12.
15. M. Stoicescu, J.-C. Fabre, and M. Roy, "From design for adaptation to component-based resilient computing," in *PRDC*, 2012, pp. 1–10.