Towards Dynamic Updates In AUTOSAR

Hélène Martorell, Jean-Charles Fabre, Matthieu Roy, Régis Valentin

Towards Dynamic Updates In AUTOSAR

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

[1] RENAULT Technocentre, Guyancourt, France
{helene.martorell, regis.valentin}@renault.com,
[2] CNRS, LAAS, F-31400,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. Vehicles are nowadays increasingly software-intensive.Current practice shows that the whole software system is tested, validated and then uploaded in ECUs in a monolithic fashion, before a vehicle is put in operation. Yet, adding specific features or updates without restarting the full validation and upload procedure could yield to significant gains. Unfortunately, the AUTOSAR (AUTomotive Open System ARchitecture) standard does not include native support for software updates. In this paper, we define a set of new concepts in order to support dynamic software updates for automotive systems, and present mechanisms that implement these concepts within AUTOSAR.

1 Introduction

When developing an automotive application, validation, testing and upload are performed comprehensively: the code is uploaded at once in the ECU (Electronic Control Unit) and when set up in the vehicle the code cannot be easily modified. Indeed, the whole compilation chain has to be restarted, therefore modifications are expensive and techniques to perform them are not standardized.

For this reason, we aim at allowing partial updates of automotive software in ECUs. This technique could enable either an upgrade of current functionalities, or addition of new ones. This would result in a gain of time and money. Indeed, all along the lifetime of the vehicle partial update make sense. Firstly on the assembly line only core functionalities could be loaded in the ECU, other functions being considered as options the client could load. Secondly, during the lifetime of the vehicles, the car manufacturer could insure the software is up-to-date without calling vehicles back to the garage. Finally, logistics and required material are reduced with this approach.

Nowadays, AUTOSAR [4] (Automotive Open System Architecture) is the standard for automotive software systems. It is a component-based layered architecture that aims at ensuring compatibility between software coming from different suppliers. Enabling dynamic software updates in this context is a major challenge since the architecture is frozen before compilation, and does not

enable any further update by design. Thus, we seek at defining a high level model of AUTOSAR to specify *adaptation areas* that will enable to fit in as many different functions as possible. These carefully chosen adaptation areas will then correspond to placeholders, or *containers*, added in the application and which can later be filled in with new functionalities or be used for updates.

The paper is organized as follows: Section 2 summarizes shortly AUTOSAR concepts necessary to our methodology. We then present the overall approach itself in Section 3, provide details of our design for adaptation and a case study in Section 4. Finally, we compare our approach with related works in Section 5 and conclude.

2 Context

This section sums up briefly the different concepts within AUTOSAR required to define our methodology. We assume the reader has some familiarity with AUTOSAR.

AUTOSAR is a layered architecture, divided into four levels. The bottom layer corresponds to the *hardware layer*. Above this, stands the *basic software layer* that contains low-level services and the operating system. This layer does not only contain standard elements, but also a number of ECU-specific components. Between the basic software layer and the application software components, the *Run Time Environment (RTE)* acts as an ad-hoc middleware. The RTE handles communication between software components and between software components and the basic software. Finally, the *applicative layer* contains specific components, which are unaware of lower layers, and that implement functions. Thus most of the components may be reused more easily on different targets (with the exception of sensor and actuator components).

From the different AUTOSAR concepts, we only concentrate in the remainder of this work on:

- RTE (Run-Time Environment): glue code that handles the communications inside the ECU
- SWC (Software Components): Application functions made of *runnables*
- Runnables: pieces of code actually implementing functionalities
- Communication: Implicit —read at the beginning, written at the end of an execution instance—, or Explicit —read when needed, written when produced— data that pass through the RTE
- OS task: Each runnable needs to be mapped into a task of the OS (AUTOSAR OS) for executing.

3 Overall Approach

Dynamic updates are not supported in AUTOSAR specifications: everything must be defined before the RTE is generated. Therefore, the degree of freedom necessary for allowing dynamic updates must be studied and integrated into

the architecture upstream. The update should not require to reload the whole application, but enable to modify some functions or add new ones in a methodical way.

AUTOSAR corresponds to a Top-down approach, and in order to comply with the requirements of the standard, dynamic update mechanisms have to be modelled and integrated. In this work, we only focus at the moment on the application layer. This means that this approach will only allow the addition or upgrade of high level functionalities.

It is important to take into account the key elements at design and development levels of an application to determine adaptation spaces. Focus is on updates for the application software, i.e., SWC that are above the RTE. Thus, this means that either a SWC or a runnable can be updated. However, a Software Component is a group of runnables that has no real existence in the final implementation. Therefore, a desirable granularity for update is the runnable. Notice that this also allow for updating a full software component by updating all of its runnables at once.

Fig. 1 shows part of a V-shaped development cycle for a standard application, and describes, on three different steps of the left-part of the V-cycle, the different AUTOSAR concepts we focus on. The general conception level corresponds to designing the whole application with its tasks (for mapping the runnables), data and runnables (which are then grouped into SWCs). Then we focus on each runnable and its specific characteristic and finally the application needs to run in an ECU where implementation specifications will matter.

In a nutshell, a set of possible values on the different axis presented on the radars in Fig. 1 defines the edges of adaptation areas that will fulfil corresponding and desirable properties. Based on these adaptation areas, containers are implementations of such areas and allow for storing and executing any new runnable whose characteristics are contained in the adaptation area.

The next part details the design of an application for adaptation. Indeed, existing automotive applications need to be prepared off-line in order to accept dynamic updates.

4 Design for adaptation and example

To begin with a pre-wired approach was developed: mechanisms and degrees of freedom added in the application are studied and integrated beforehand. The level of granularity chosen is a runnable (the radar presented in the red rectangle on Fig. 1). On this radar, values can be allocated on each axis in order to define an adaptation area. Then based on this area a *container* can be added upstream in the application: it corresponds to a placeholder that presents the correct properties w.r.t. a given adaptation area.

In order to determine the most relevant adaptation areas, the most frequent runnables in automotive systems were studied. We analysed a few automotive applications used by Renault. On average 70% of tasks are periodic and 71% of

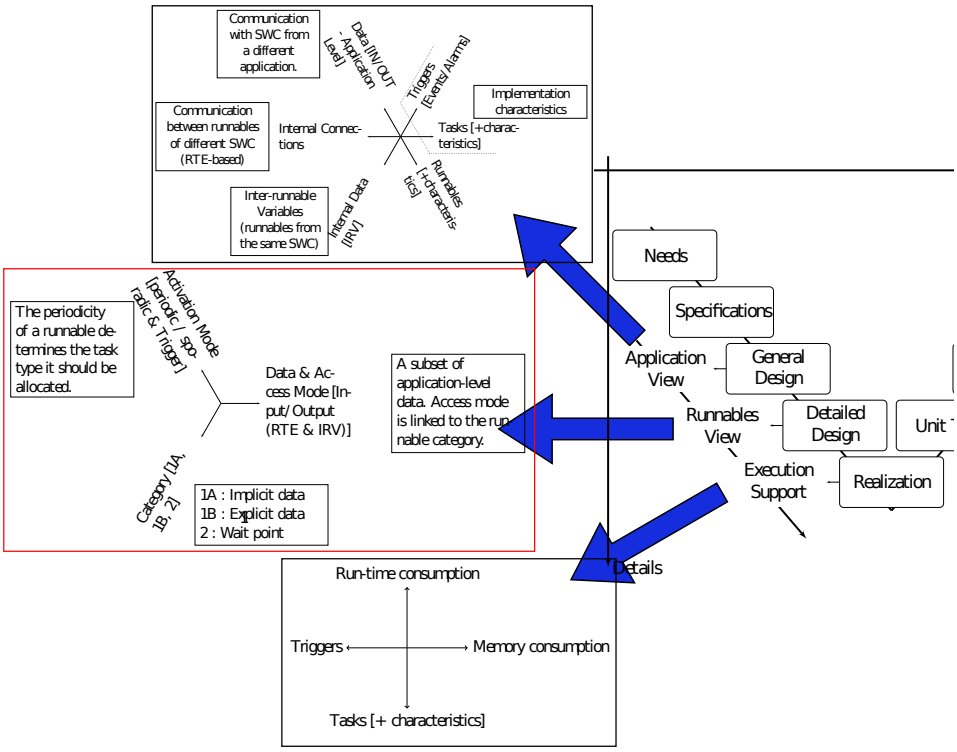

Fig. 1. Development cycle with corresponding potential adaptation spaces

all runnables are mapped in these tasks. The other tasks typically either execute only once on start-up or are event-triggered.

Communication-wise, considering the huge amount of possible data in an automotive application, the runnable will reuse existing channels in the RTE in order to communicate with its environment. Indeed, when the RTE is generated, an API that defines all the necessary communication is created. Therefore, the runnable can reuse this API for fulfilling its communication needs.

In this context, to increase modularity, the addition of an extra level of indirection between the runnable and the caller is necessary. This can be realized by slightly modifying the AUTOSAR process: an extra step in the tool-chain [9] has to be added to automatically modify the calls to runnables for adding an extra level of indirection. In the AUTOSAR process, the different SWC can be delivered either as source code or object code. The extra step in the process should be able to handle this specificity.

To support our tests, we used an operating system compliant with AUTOSAR OS specifications: Trampoline [1]. It is an open source OSEK/VDX OS implementation which enables testing on different embedded targets as well

as on POSIX. This latter feature is the one we used to start with in order to experiment with our methodology.

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 sensor 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.

For the update, we first created an empty container, and an update runnable that makes it possible to have a twice as fast blinker. Hence, filling up this container is easily seeable. Moreover, this application contains a variety of tasks and runnables which can provide support for different kinds of adaptation areas.

The first tests we carried out were on a POSIX target, that enables to show the feasibility of the approach, but does not provide realistic timing and memory consumption measurements. We started with simple tests for verifying our approach: runnable update, container creation, uploading and execution of the new runnable in the container. Using a tool we developed for automatically adding indirections and `gdb` for testing and debugging purposes we were able, on the POSIX target, to interrupt the regular execution of the program for verifying the proper behavior of updates that were introduced in the applications.

The second step for the tests was to upload the application in an actual automotive ECU and carry out simple tests which produce visual outputs on the blinkers. This enable for some performance measurements. We focus here on memory and run-time impact. Memory wise, an update manager and charger must be added: this represents an increase of around 15% in the original memory footprint. At run-time, the only time overhead corresponds to a dereference operation of a pointer which can be considered as negligible.

Note that in this paper the concern about safety for the update of critical functionalities is not addressed. However, an analysis of the undesired events that could be caused by the updates need to be performed and appropriate safety mechanisms added.

5 Related Work

McIllroy was the first to describe component based architectures in 1969[2] 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, and increases the quality and evolution of the software [3].

For partial update of component-based software, there are two major approaches: routine-based update and component-based update.

Routine-based update corresponds to a finer granularity as it typically updates individual functions or objects. For example Ksplice [5] or Ginseng[6] use a system of patches for hot updates without reboot, and replaces entire functions. This can be assimilated to the update of a runnable, but their approach does not focus on embedded systems.

Component-based update also corresponds to our work : updating several runnables corresponds to updating a SWC. Previous work in this domain are for example Li *et al.* in [7]. They present an OSGI-based automotive specification: a component-based platform that enables to download, install and uninstall bundle(service application). Another component-based approach for embedded systems is described in [8]. Although both these approaches focus on embedded systems they are not compliant with the AUTOSAR specifications.

6 Conclusion

We presented here the model we built for designing adaptation areas and their associated containers, which are implementation counterparts of adaptation areas. 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. These containers must have specific features that will have match future runnables.

We focused on concepts that are necessary for dynamic update and a model for empty containers for allowing posterior dynamic updates. The adaptation engine that will perform the actual update is beyond the scope of this paper.

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. One of the future directions we investigate is to add safety mechanisms to ensure dependability of dynamic updates [10].

References

1. J.- L. Béchennec, M. Briday, S. Faucou and Y. Trinquet, *Trampoline - An Open-Source Implementation of the OSEK/VDX RTOS Specification*, ETFA 2006
2. M.D. McIlroy.*Mass produced software components*. In P. Naur and B. Randell, editors, Software Engineering, pages 138–150. NATO Science Committee, Januar 1969.
3. Crnkovic, Ivica, *Component-based software engineering new challenges in software development*, Software Focus, vol 2, John Wiley and Sons, Ltd., 2001
4. AUTOSAR, http://www.autosar.org/
5. J. Arnold and M. Frans Kaashoek. 2009.*Ksplice: automatic rebootless kernel updates*, 4th ACM European conference on Computer systems (EuroSys '09), New York, USA
6. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. *Practical Dynamic Software Updating for C.*, PLDI, June 2006.
7. Y. Li, F. Wang, F. He, Z. Li, *OSGi-based service gateway architecture for intelligent automobiles*, Intelligent Vehicles Symposium, 2005.
8. M. Wahler, S. Richter, and M. Oriol. 2009. *Dynamic software updates for real-time systems*. HotSWUp 2009. ACM, New York, NY, USA
9. S. Voget. 2010. *AUTOSAR and the automotive tool chain*. DATE 2010. European Design and Automation Association
10. T. Piper, S. Winter, P. Manns, N. Suri, *Instrumenting AUTOSAR for dependability assessment: A guidance framework*, DSN 2012