Improving Adaptiveness of AUTOSAR Embedded Applications

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

Improving Adaptiveness of AUTOSAR Embedded Applications

Hélène Martorell[1][2][3] Jean-Charles Fabre[2][3] Matthieu Roy [2][4] Régis Valentin[1]
{helene.martorell,regis.valentin}@renault.com {fabre, roy}@laas.fr

[1]Renault S.A.S.
1, avenue du Golf
Guyancourt, France

[2]CNRS, LAAS
7 avenue du colonel Roche
F-31400, Toulouse, France

[3]Univ de Toulouse
INP, LAAS
F-31400 Toulouse, France

[4]Univ de Toulouse
LAAS
F-31400 Toulouse, France

ABSTRACT

AUTOSAR (AUTomotive Open System Architecture) is the most recent standard for automotive embedded systems. A major drawback of AUTOSAR lies in its lack of flexibility. Software-wise, ECUs (Electronic Control Unit) are tested, validated and uploaded; these three steps are performed in a monolithic process. Adding *adaptability* features into the standard is a major challenge in this context, and may result in consequent savings of time and money. On-line adaptation allows the inclusion of new functionalities in an efficient way. In this paper, we first extract relevant features from AUTOSAR for allowing dynamic software updates. Then, we define our approach for performing updates, and provide an evaluation of our approach on a RISC micro controller.

Keywords

Partial Updates, Architecture, Automotive Embedded Systems, Real-Time System, AUTOSAR

1. INTRODUCTION

Automotive embedded systems are usually characterized by scarce resources and a real-time behaviour. Custom designs were therefore preferred for optimization purposes. Yet, growing complexity and increase number of partners led to the creation of a standard called AUTOSAR [2] for improving time-to-market and reducing costs. AUTOSAR defines a component-based architecture with standardized interfaces[6] for increasing reuse of components by abstracting from the hardware.

Once the ECU (Electronic Control Unit) is set up in the vehicle, no standard ways for updates currently exist: either the ECU is completely reloaded or low level patches are set up. Therefore changes are, at the moment, quite expensive and done in an ad-hoc way. In this context, allowing partial updates of automotive software is a hot topic [11][8]. In our view, this encompasses both upgrades of current functionalities and addition of new ones. Such a mechanism would, for example, allow the addition of functions that were not available when the vehicle was first put into service.

Partial updates within AUTOSAR – concepts.

Enabling partial software updates in AUTOSAR is a major challenge that we tackle in this paper. AUTOSAR imposes a static and frozen architecture, and thus disables by design any further update. Thus everything must be defined before compiling and the approach we offer is static: everything is defined beforehand. A *high level model* of the software architecture, compatible with AUTOSAR, is designed to specify *adaptation areas*. The latter will enable to fit in *new or evolved functionalities* during system lifetime, and as such, should contain all relevant information. Obviously, in order to be acceptable by the automotive industry, the process should be fully compliant with the AUTOSAR standard. Our main motivation in this paper is to define a clean and standardized way for allowing partial updates and upgrades within the boundaries defined in the standards.

Off-line support.

The second step to perform updates, once adaptation areas have been carefully defined, is the development of their implementation counterparts. The realization of these placeholders, that we call *containers* in the remainder of the paper, have to be added to the application before compilation time. These container can then be filled during the operational lifetime of the system, either with a new implementation of an existing functionality or a new functionality altogether. The creation of updates also need specific off-line support, in particular for testing and validation purposes. A framework should be available during the lifetime of an application for developing these updates.

Runtime support for partial updates.

The updates need to be first tested offline for validation. Loading an update is done by the *update manager*, which checks whether a given update can fit (at least) one free container. The update manager also has to perform several compatibility checks. Then, during the lifetime of the newly added components, online safety protections should be added. Those last two points are mentioned here for the sake of completeness. Potential solutions to implement such protection mechanisms are based on runtime assertions and safety bags [5]. In this paper, we focus on designing the implementation of the adaptation features according to AUTOSAR constraints.

The paper is organized as follows: Section 2 describes the context of our work and presents briefly the AUTOSAR standard, extracting key elements to define our approach. We then introduce the overall approach itself in Section 3, provide details of our design for adaptation in Sections 4, 5 and 6. We describe an early proof of concept and related tests we performed in Section 7. Finally, we compare our approach with related works in Section 8 and conclude.

2. CONTEXT

2.1 Motivation

As for most critical systems, automotive embedded systems are extremely static, and introducing a way for allowing updates is not a straightforward process. Yet, dynamic updates are an upcoming and necessary feature for vehicles.

Currently, adding features that were not available when the vehicle was put in service is not possible. Therefore car owners cannot keep up with the latest technology and their vehicles can become outdated. Enabling updates in AUTOSAR ECUs could allow them to benefit from software technology that were developped after they purchased their vehicles.

Furthermore, when buying a new car it could become possible to customize vehicle at a fine-grain level easily by allowing customers to select the software options they value most.

This could later result in some optimization during the manufacturing process of vehicles: the amount of software uploaded can be reduced for minimizing the corresponding latency time. In such a scenario, only core functions of the software would be loaded, and everything that is considered as an option can be set aside at that time. By extension, car realignments (i.e the reprogramming of an ECU between initial load and shipping to customer) would become less expensive since only the changes would have to be loaded.

Maintaining a vehicle up-to-date during its lifetime could also be eased by allowing the car owners to perform updates without the constraint of going back to the garage.

Finally, partial updates would also reduce logistic as they require less material, and be significantly quicker than reflashing the whole ECU —experience shows that we can have transfers up to 50 times faster since the amount of data to transfer is significantly smaller.

2.2 AUTOSAR concepts

This sections gives an incomplete presentation of the standard. It only aims at identifying the different dimensions of an AUTOSAR application which will be necessary for modelling purposes and for defining a reference for adaptation.

AUTOSAR is a layered architecture divided into four main levels. The bottom layer corresponds to the *hardware*. Above the latter stands the *basic software* layer that contains low-level services and the operating system. The top layer corresponds to the applicative layer divided into specific software components. The latter are unaware of lower layers, and implement functions. Finally, between the basic software layer and the application layer, the *Run Time Environment (RTE)* acts as an ad-hoc middleware.

2.2.1 Run-Time Environment (RTE)

The RTE enables the software components to communicate with one another and with the basic software. Using specific tools, the RTE is automatically generated to conform with application's specifications. Its roles are to handle communications within the ECU and trigger the execution of functions by sending events. Eventually RTE can be seen as a collection of communication channels either between different software components or between components and basic software. A communication channel enables an amount of data to be sent (resp. received) by a SWC to (resp. from) another SWC or an element of the basic software. These are static: source and destination must be known at generation time, although the SWCs themselves do not the identity of the other SWCs it communicated with (this improves reuse).

2.2.2 Software Components (SWC)

Software Components (SWC) correspond to application functions. A SWC is defined as a group of fragments of executable code that are called *runnables* (a sort of processing step, e.g. a C function). A runnable implements a specific function and can be executed either periodically or on the occurrence of a specific event (e.g. the reception of input data). A runnable can also wait for an external event (at a wait point). The presence or absence of wait point divides the runnables into two categories: Cat. 1 corresponds to runnables without wait point and Cat. 2 to runnables with wait point. The rationale behind this classification is first to separate runnables for which execution will surely terminate (Cat. 1) from those that could hang (Cat. 2), e.g. when waiting for user's input. In order to communicate with other SWCs or with the basic software layer, a SWC is associated with input and output ports.

Notice that SWCs only correspond to a structural breakdown that does not have real existence in the final binary objects. Indeed, this role is played by the *runnables* that are mapped onto *tasks* of the operating system, regardless of which SWC they belong to. Thus, *runnables* and *tasks* are key concepts that must be considered in our approach for adaptation.

2.3 Operating System principles

The operating system in an automotive context essentially deals with the scheduling of tasks, alarms and events. AUTOSAR OS is the real-time OS associated to the AUTOSAR standard. It implements the following main characteristics: fixed priority scheduling, interrupt handling, and protection against unintended uses of OS services.

2.3.1 Tasks model

Runnables of SWCs need to be mapped onto tasks, thus the task model is an important aspect of our analysis. There are two types of tasks in AUTOSAR OS: basic tasks and extended tasks. The main difference between them is that an extended task can wait for an event while a basic cannot. Thus basic tasks only get synchronized at the beginning and at the end of their execution while extended tasks enable more synchronization points since they can interact with other tasks using events.

2.3.2 Allocating runnables onto tasks

When creating an AUTOSAR application all the runnables belonging to the various SWCs must be mapped onto tasks for execution. Runnables can be seen for example as functions in C which will need to be activated and scheduled by the OS. All runnables are called within the body of various tasks.

The allocation is usually done by the integrator of the system. It depends on the available tasks and the needs of the runnable, e.g. periodic runnables need to be put into tasks either with the same period, or with a smaller period. If the second option is chosen, then the runnable will not be executed each time the task executes: glue code will be automatically added at generation time so that the runnable only executes at the appropriate period. On the

other hand, aperiodic runnable, which would typically be event-triggered, have to be mapped onto extended tasks (see section 2.3.1).The code for each task's body with the proper glue code for tasks execution and each runnable called in order is generated.

3. APPROACH FOR PARTIAL UPDATES

Partial updates done dynamically are not supported in AUTOSAR: everything must be defined when the RTE is generated. Therefore, the degree of freedom to be integrated into the architecture for allowing this feature must be studied and integrated upstream. The goal is to load only part of the application and the modifications must be added methodically and monitored to insure proper behaviour. *Adaptation areas* are defined based on specific elements of design and development of an AUTOSAR application.

SWCs are defined in section 2.2 as a group of runnables. Thus the proper granularity for updates is a runnable: updating every runnable of a SWC means updating a SWC.

3.1 High-Level View

Application view

When designing an AUTOSAR application, firstly the necessary SWCs (and thus runnables), inputs and outputs based on the specification are determined. Communications and the characteristics of the runnables are also required, and appropriate tasks for executing the runnables are created. The different characteristics of an application are represented in Table 1. Runnables and Tasks will be detailed in the following sections: their will be the basis for defining our key concepts namely *adaptation area* and *containers*. Data, on the other hand, are relevant to determine which communication channels will be already available for the future updates.

Table 1: Characteristics of AUTOSAR Applications

	Details	Comments
Tasks		Tasks and corresponding characteristics
Runnables		Runnables and corresponding characteristics
Internal Connections	Communication between runnables of different SWC	Communications within one ECU using the RTE.
Data [IN/OUT]	Input and Output data Application-wise	Communication with SWC located on different ECUs (use of communication bus)

Runnable View

Table 2 shows the relevant characteristics of a runnable w.r.t. partial update, i.e. activation mode, Inputs and Outputs and Category (1, 2). The key characteristics here are *Activation Mode* and *Category* as data depend on the functionality of each runnable.

3.2 Execution support

To upload a complete application with all its characteristics (Table 1) in an ECU embedded into a car, it requires to take into account the execution support. From the OS point of view, execution support means tasks (with their features). The characteristics for these tasks are presented in Table 3. The memory consumption of the application and the time used for execution are also important parameters

Table 2: Characteristics of AUTOSAR runnables

	Details	Comments
Activation mode	Periodic / Sporadic	A runnable is either periodic or triggered by events.
Data	Input and Output Data (RTE-base and IRV) and corresponding access mode	Corresponds to communication needs of the runnable. Required to plan ahead the proper connections. Linked to the runnable category.
Category	1: No wait point 2: Wait Point	A wait point is the moment when a runnable wait for an external event to resume its execution

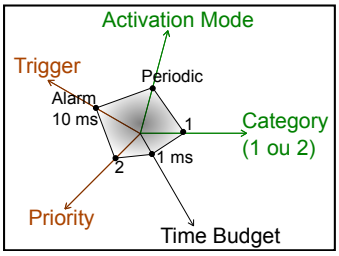

Figure 1: Example of an adaptation area

which need to be taken into account. Yet, these parameters are target specific.

Definition (Adaptation area) A set of possible instantiations of different parameters extracted from Tables 2 and 3 defines *the edges of adaptation areas* for an update runnable. Figure 1 shows an example of such an adaptation area and the specific characteristics. Activation Mode and Category (green) axes are extracted from Table 2 and Trigger and Priority (brown) ones from Table 3. An extra axis is added for taking into account timing characteristics.

In an AUTOSAR development process runnables are defined first with their corresponding characteristics, and tasks must be then created to satisfy these characteristics. For this reason, tasks characteristics encompasses all the characteristics of a runnable in addition to their own.

Table 3: Characteristics of AUTOSAR OS Tasks

	Details	Comments
Type	Basic / Extended	A basic task cannot wait for an event, only extended tasks can.
Activation Mode	Periodic or Event-triggered	Periodic tasks are triggered by alarms and sporadic tasks by events.
Preemptive	Full-preemptive Non-preemptive	Suspended if a higher-priority tasks is ready or by an interrupt Only suspended by an interrupt
Priority		A higher number means higher priority.
Trigger	Alarm Events	Periodic Tasks Sporadic Tasks
Activation mode	Periodic / Sporadic	

4. ADAPTATION AREAS & CONTAINERS

The granularity for updates being the *runnable*, an adaptation area will correspond to all possible characteristics for fitting a runnable within the application on an ECU. It is represented by a combination of values for the different possible parameters in order to outline an area in the application with desirable features (see example on Figure 1).

Definition (Container) A *container* is a physical representation of an adaptation area that will implement the required characteristics in order to act as a placeholder in the embedded application. It is designed for harboring an update runnable with matching characteristics. Figure 2 shows an example of a container placed inside a task.

In this work, we use a pre-wired approach: the level of adaptation available for a given application is determined beforehand and prepared off-line. Both structural and run-time characteristics of the containers must be determined when they are placed in the application.

When a container is created, it is allocated a time budget — maximum amount of time available for code that would be set up in it. However, when any container is empty, it only corresponds to the execution of an empty function: no time is used; the extra time added for the containers will be considered as slack time. When an update occurs and a runnable must be placed inside the container, the WCET (Worst Case Execution Time) of the update runnable should not be greater than the time budget of the container.

When adding a container corresponding to an adaptation area in the application, a suitable task needs to be found. This task should have matching characteristics (Table 3), in particular regarding its type and activation mode (periodic or event-triggered). Obviously, the scheduling analysis should take into account the new container's time envelope or, to be more specific, the WCET any update loaded into the container must satisfy. If all these criteria are met, it is then possible to add this specific container into the application.

An example of a task with an empty container is shown on Figure 2. On this example the main rectangle represents the task, the smaller one either runnables or containers and the arrows stand for communications.

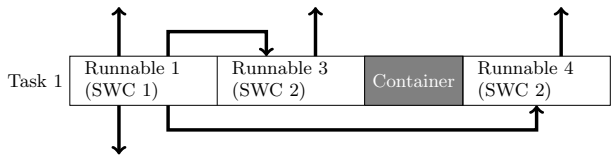

Figure 2: Example of task with an empty container

When designing adaptation areas and consequently containers, their characteristics must be carefully chosen. They have to implement a trade-off between *genericity* (being general enough to support the fitting of as much runnables as possible) and *performance* (all resources used by the empty container are "wasted" until the container is filled).

Therefore, we base our implementation choices on recent statistics on real automotive systems: the kind of runnables that are the most frequent in automotive system should give a good indication of parameters for adaptation areas. To select the appropriate adaptation area, characteristics of the needed containers have to be defined along the various parameters of the reference model. Our analysis is based on the study of a few automotive applications used by Renault. On average, 70% of tasks in these applications are periodic. The remaining 30% are typically tasks that either execute only once on start up or are event-triggered. Furthermore on a runnable level, 71% of runnables are periodic. This means that to start with, it is meaningful to add containers in periodic tasks. Note that the percentage of periodic runnable is independent from the number of periodic tasks since several runnables can be mapped onto a task.

5. PROCESS BASED CODE GENERATION

The general process for developing automotive software in compliance with AUTOSAR standard is shown on Figure 3. The top level corresponds to the early stage of the design and the bottom to the actual code.

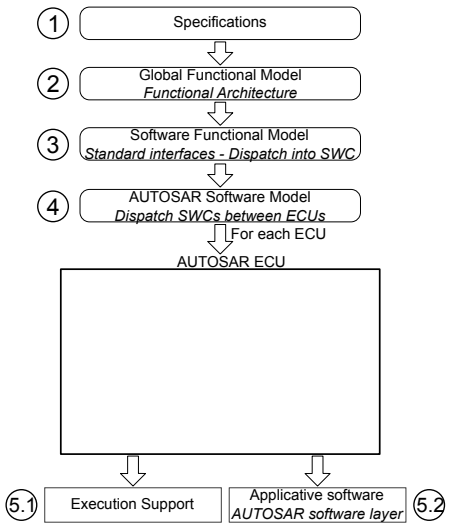

Figure 3: Automotive embedded software Process

Starting with the specifications (step *1*, the different high level functions, necessary for fulfilling the needs, are created: this is the *Global Functional Model* (step *2*). Once all the functions are available, they are divided into Software Components to create the *Software Functional Model* (step *3*). If amongst the different functions that have to be implemented, several of them are common, or have similar purposes, they can be grouped inside a same SWC regardless of the initial high level function it belonged to. The communication channels also have to be defined at this point. This is a high level view of the communication channel: it does not matter to which ECU the SWC will be allocated. Then the various SWC are dispatched between the different ECUs (step *4*). This creates a system that is distributed amongst several ECUs. Yet, in this work, we focus on a single ECU.

Finally, each ECU has to be processed in two parallel steps. Firstly, the lower level layers (step *5.1*): the RTE, the OS and MCAL and the Basic Software. This is AUTOSAR-specific glue code for the RTE and hardware specific functions that are necessary for the proper execution of applicative code. Then the functional code of the application is created (step *5.21*).

6. DESIGN FOR ADAPTATION

6.1 Process Modifications for Updates

The process described in section 5 needs to undergo two slight amendments to support partial updates. Firstly, in the early stages of development for automatically adding *containers* in the original *Software Functional Model* (at step *3*). Among the software components, one or more hollow ones are added: each of their runnable will correspond to the envelope of containers. Adding them early in the process allow us to pass them along and propagate the changes.

These empty functions are then passed to the *AUTOSAR Software Model* (step *4*), and each of them corresponds to a SWC with containers (i.e. the representation of a number of adaptation areas). Finally these containers are set up in existing tasks matching their characteristics.

The second amendment corresponds to a post-processing step on one of the RTE files (step *5.1*). Indeed, it is necessary to add all the necessary mechanisms for handling the updates: an extra level of indirection in order to modify the execution flow and run the new updates and an update manager for handling the changes. These post processing operations have been automated using python scripts.

6.2 Creation of an Update Runnable

Prior to their integration into an application with the appropriate update services, *Update Runnables* undergo a design process. The latter is shown on Figure 4, from the creation of an update to its integration. Updates are first tested off-line: starting with unit tests that check that the required functional properties are properly implemented. Then integration tests are carried out: the update is tested within a representative context. The update runnable has to be tested off-line within the AUTOSAR application it will later be integrated to. In particular, the execution flow, the scheduling and the communication flows. This last point is particularly relevant since the runnable will reuse existing channels in order to communicate with its environment.

After tests completion, the runnable are validated within the application. This way, when the runnable is validated, it can be integrated to the embedded application. The *Update Manager* will handle the update runnable in order to load it inside a container. On the left hand side Figure 4 shows the process for creating the update runnable. On the right hand side it shows the current AUTOSAR application with update services, containers, and points in the application pointing to the container for executing the updates.

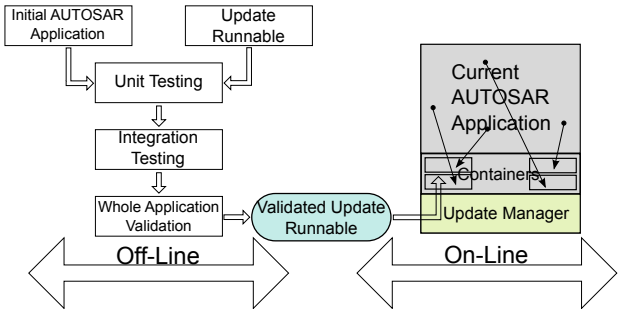

Figure 4: Integration of an update runnable

6.3 Technical Approach

There are several steps which must be performed to upload the runnable into a container and ensure it executes correctly in the current environment. Notice that the new runnable is tested, in its operational environment, off-line by the system manufacturer (see section 6.2. However, some subtle runtime parameters can have an impact on its behaviour on the target system. This calls for defensive integration based on run-time assertions, e.g., as a wrapper for dependability.

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 [13] has to be added to automatically modify the calls to runnables and add this extra level of indirection. This kind of process was used for safety purposes [10] and dependability properties [7].

In the AUTOSAR process, SWC can be delivered either as source code or object code. The functional code of SWC might not be available for adding indirections. Yet, Figure 3 shows that the RTE is generated independently from the actual code of the SWC in the AUTOSAR related branch of the process; it is a reasonable hypothesis to consider that the source code of RTE will be available. Thus, we can modify directly the RTE code in order to add a new level of indirection. This step is performed thanks to a tool called *Indirection Handler* that we developed —*cf.* Section 6.5.

6.4 Necessary Meta Data

Along with the generation of code some meta data are created, some of which are necessary for the dynamic updates, e.g. the address of each runnable in the ECU. This is necessary when an update occurs in order to swap the appropriate runnable. The table corresponding to our extra level of indirection is also necessary for modifying it.

For safety purposes it is also important to offer a possibility for rolling back to a previous version (e.g. if a problem occurs with an update or an upgrade). A support for restoring the system into its initial configuration is also a desirable feature. This requires storing the initial address along with the current runnable address. Meta data, that will enable to keep the values for a default configuration or a previous one that is guaranteed to work properly, must be stored.

Finally the address of every container must be stored along with a corresponding descriptor for determining their characteristics, verifying they are empty and filling them.

6.5 Indirection Handler

To instrument the application and add the necessary extra level of indirection, relevant characteristics of runnables must be extracted from the description file corresponding to each SWC. Tasks' bodies are then modified accordingly. This corresponds to a post-processing of the RTE files. This step needs to be done automatically for a better integration to the process using a python-based tool we developed.

The *Indirection Handler* is divided into two steps. The first step consists in extracting relevant information for adding an extra level of indirection from the `.arxml` description provided with each SWC, i.e. necessary information relative to the runnable (name, period, etc).

When runnable information is available, the second step modifies the `Rte.c` file. This file contains the body of all tasks in the system and therefore the direct call to each

runnable mapped into them. These calls are modified for actually adding the indirection level along with specific mechanisms for a subsequent dynamic update.

7. IMPLEMENTATION AND TESTS

The objective here is to create and execute a simple automotive software that includes containers and update mechanisms. Containers can then be filled in with appropriate update runnables, during the lifetime of the system. It is necessary to prepare the application before compile time: we add indirections and mechanisms to enable a switch from an empty container to an actual runnable.

To support our tests, we used an operating system compliant with AUTOSAR OS' specifications: Trampoline [3], an open source implementation.

7.1 Simple example application and tests

For testing purposes, a simple application was extracted from an ECU similar to BCM (Body Control Module) and modified. This application is used as an early proof of concept for showing the feasibility of updates within AUTOSAR. It controls the blinkers in a car by reading from the sensors (turn switch sensor and warn light sensor), treating the received data and sending a signal to the actuators to trigger the flashing of the light bulbs.

A main motivation for choosing this application is its visual feedback, which means that it can be put into a demonstration board and *anyone can observe* the outputs of this application. This feature cannot be taken for granted when dealing with embedded systems! For the update, we first create an empty container, and generate a software update that flashes the warnings in case of emergency braking: filling up that container will be seeable.

Experimental tests are carried out using a PowerPC processor —of type MPC5510— on a dedicated Freescale development board, along with Code Warrior IDE and compiler.

7.2 Updates and the Infrastructure

In this section we present the overhead related to the mechanisms added to the application and that is necessary for enabling updates in AUTOSAR architecture. Our main concerns here are regarding memory and run-time impact of the addition of containers and the use of indirections.

Memory is scarce in automotive systems and therefore the impact on this parameter is paramount. Our approach impacts memory on several levels. First, meta data are necessary (see section 6.4): since an extra level of indirection (see section 6.5) is added for modifying the code at run-time, each pointer's address need to be stored. Each entry in the table of indirection represents the size of an address (32 bits) and of an ID which can be the index in the table (and therefore free). For a simple application with 10 runnables and 3 containers this would represent 416 bits. This is quite small compared to the size used by the application(around 32 KB). Moreover the addresses of runnables also need to be known and placed into a table. This would also consume 32 bits per address. The size of necessary meta data will grow linearly with the number of runnables.

Table 4 shows the memory consumption for different version of our blinker application. It only shows the flash consumption since the code is stored in flash and so will be the updates. Adding a level of indirection is negligible regarding memory consumption. To have a better evaluation for the memory impact of the modules added for dynamic update, we compare the sizes to a *bare application*, more precisely an application which contains the minimum mechanisms for executing a simple runnable that periodically increments a static variable. This application is made of the necessary files for OS, basic software and specific board-related drivers. Based on this application, we add incrementally several functionalities. Adding the functional code for the blinker application represents an increase of 31.2%, adding the services for dynamic updates represents an increase of 37% compare to the simple blinkers application.

Table 4: Memory usage our test applications (Bytes)

Bare	Blinker	Blinker with indirection	Blinker with Update Manager	Blinker with Update Manager and Indirections
17748	23288	23352	31908	31972

Overall, the mechanisms for dynamic updates use *i*) a small amount of memory for keeping relevant meta data required for dynamic upload, *ii*) virtually no memory for adding a level of indirection, and *iii*) a constant amount of memory for the Update Manager (Flash Manager and Charger). Memory-wise, update mechanisms represent less than 20% of total application space for our very small example.To sum up, the bigger the application is, the smaller the memory impact of update manager represents in ratio to the total memory use.

Finally, prior to their activation, updates have to be uploaded in memory. We chose to place them right after the application, contiguously in the memory. Therefore they only use the required amount of memory; as long as there is enough memory available, further updates can be stored.

Determining the time consumption added by the runnables was done using hook mechanisms. Trampoline provides *Pre Task Hook* and *Post Task Hook* for triggering specific software instruction before and/or after each task executes. This allows for comparing execution times. We used a Mixed Signal Oscilloscope (Agilent Technologies MSO 6034A) in order to measure the execution time for each task and we take an average value on 10 executions of each task. Then we compare the application with indirections with the original one.

This early proof of concepts is a simple example with 5 tasks. Measurement shows that average run-time for each of them is identical whether the indirection level is added or not. Indeed, this only corresponds to the dereference of a pointer, and is therefore negligible. Observation are only made on the periodical tasks. There are some small differences between the time spent in the tasks with and without pointer, however these differences can be explained by the measurement imprecision.We also measured the necessary time for switching from one task to another: about 9 μs.

In a nutshell, run-time performances are not impacted by the addition of the mechanisms for updates and their memory consumption is limited to around 20% of total application space. Yet, the updates themselves also use memory for storage. This fact should be anticipated, and extra memory regions have to be pre-allocated for them. The overhead for the *Update Infrastructure* is therefore limited while it brings flexibility to an otherwise fixed and frozen architecture.

Note that in the application provided by Renault Engineering, a runnable represents less than 10 kB in memory, whereas a full application represents several hundred kilo-

bytes. This means that a runnable is at least tens of times smaller than a complete application. Therefore, considering that there is some overhead related to the upload mechanisms, updating a single runnable will be at least ten times faster than uploading a complete application.

8. RELATED WORK

Adaptable software will enable cheaper maintenance, better ability to cope with complexity, to increase the quality and evolution of the software [4].

For updating component-based software, there are three major approaches: routine-based update, component-based update and updates at the granularity of the whole program.

Routine-based update corresponds to a finer granularity as it typically updates individual functions or objects. For example Ksplice [1] 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 toolchain[13]: dynamic update will require an extra step.

In [14], 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. Several techniques aiming at dynamically updating component-oriented embedded system where studied and compared in [12]. Yet, none of the proposed technique is designed for automotive system.

9. CONCLUSION

We investigated AUTOSAR-related concepts required for allowing partial updates or upgrades of software in an AUTOSAR application. To that aim, we presented a model we built for designing adaptation areas and associated containers, which are implementation counterparts of adaptation areas. Design for adaptation, application of the concepts to an early proof of concept were also detailed in the last sections of this paper. In this work, we use a "pre-wired" approach: we define at design time adaptation containers that can afterwards be filled in with new runnables.

These containers must have specific features that will correspond to future runnables'. The impact of our approach on run-time and memory consumption has been studied on a simple yet real automotive application. Interestingly, the run-time impact of the mechanisms is negligible from the timing point of view since it solely consists in adding a level of indirection. Moreover, since the system is designed for accommodating new runnables, added updates can be taken in by the currently running application without creating execution overhead. Memory consumption is also limited since specific services have a fixed size and necessary meta-data for performing updates grows linearly with the application. However, adaptation is not free as extra space has to be allocated beforehand, both for memory and execution time.

In the future, our adaptation engine should handle several extra contingencies such as keeping track of available containers, or detecting new updates available.

Dynamic updates must be monitored for safety purposes: the update should not prevent the system from working properly. We are investigating techniques to add safety mechanisms to ensure safety of dynamic updates [10, 7]. Indeed, one of the interesting use of dynamic update could be to quickly spread bug-fixes on a large scale.

10. REFERENCES

[1] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. ACM EUROSYS*, pages 187 – 198, 2009.

[2] AUTOSAR Development Cooperation. www.autosar.org.

[3] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline - an opensource implementation of the OSEK/VDX RTOS specification. In *IEEE EFTA*, 2006.

[4] I. Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001.

[5] J.-C. Fabre, M.-O. Killijian, and F. Taiani. Robustness of automotive applications using reflective computing: lessons learnt. In *SAC*, pages 230–235. ACM, 2011.

[6] S. Furst, J. Mossinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkamper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR - a worldwide standard is on the road. International VDI Congress Electronic Systems for Vehicles, 2009.

[7] C. Lu, J.-C. Fabre, and M.-O. Killijian. An approach for improving Fault-Tolerance in Automotive Modular Embedded Software. In *RTNS*, 2009.

[8] 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/.

[9] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[10] T. Piper, S. Winter, P. Manns, and N. Suri. Instrumenting AUTOSAR for dependability assessment: A guidance framework. In *DSN*, pages 1–12, 2012.

[11] R. B. Software. Red bend partners with vector to update automotive ECUs using delta and over-the-air technology. June 2013. redbend.com/?option=com_releases&view=article&id=1976.

[12] B. Y. Vandewoude Yves. An overview and assessment of dynamic update methods for component-oriented embedded systems. In *ICSM*, pages 521–527, 2002.

[13] S. Voget. AUTOSAR and the automotive tool chain. In *DATE*, pages 259–262. IEEE, 2010.

[14] M. Wahler, S. Richter, and M. Oriol. Dynamic software updates for real-time systems. HotSWUp '09, pages 2:1–2:6, New York, NY, USA, 2009. ACM.