Partial Updates of AUTOSAR Embedded Applications - To What Extent?

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

▶ **To cite this version:**

Hélène Martorell, Jean-Charles Fabre, Michaël Lauer, Matthieu Roy, Régis Valentin. Partial Updates of AUTOSAR Embedded Applications - To What Extent?. 11th European Dependable Computing Conference (EDCC 2015), Sep 2015, Paris, France. hal-01194832

HAL Id: hal-01194832
https://hal.science/hal-01194832

Submitted on 7 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Partial Updates of AUTOSAR Embedded Applications — To What Extent?

Hélène Martorell*§, Jean-Charles Fabre*, Michael Lauer*, Matthieu Roy* and Régis Valentin§
* LAAS-CNRS, Univ. Toulouse, France
Email: {martorell|fabre|lauer|roy}@laas.fr
§ Renault T.C.R., Guyancourt, France
Email: regis.valentin@renault.com

***Abstract*—The AUTOSAR standard describes an architecture for embedded automotive systems. The lack of flexibility is a major drawback of this architecture and updates are not easily possible. In our work we explore the various facets of software updates in the context of AUTOSAR embedded applications. With few modifications that remain compatible with the development process, we propose specific concepts for updates. Such updates can be remotely achieved, for maintenance and/or evolution purposes. As functional updates may lead to safety mechanisms updates, we also highlight how safety mechanisms can be added or updated with different level of granularity. We illustrate these concepts and capabilities with a simple case study as a proof of concepts. We finally draw the lessons learnt from this work.**

I. Introduction

AUTOSAR (AUTomotive Open System ARchitecture) is a standard defining a software-based architecture for automotive embedded systems [1]. The main benefits of AUTOSAR rely on standardized interfaces, ease of maintenance, simpler reuse and integration of software components. Nonetheless, a major drawback of this standard is its lack of flexibility. The complete configuration of the system is usually done before compile time and almost no modification is possible afterwards. Therefore, a traditional way for updating automotive embedded software is to reload completely the ECUs (Electronic Control Units).

Remote partial updates would allow for easier improvement of the software within vehicles. The idea is to modify or add tiny software entities in an ECU, e.g. the modification of an existing functionality may lead to update a small function, namely just a few kilobytes. Adding dynamicity would allow a much easier and remote maintenance for the car owner who would not necessarily need to go back to the garage. It can also help taking advantage of the latest improvements and technologies: a software option that was either not available or not selected when the car was purchased can be added afterwards. As advocated in many talks in the automotive industry, about 90% of the functional improvements will be software-based by the year 2020. Yet, reducing time-to-market and allowing updates and upgrades of on-board software remains a challenge.

Fast adaptation of embedded automotive software is clearly very attractive for the automotive industry. The question is: *to what extent partial adaptation is possible in current AUTOSAR-based application whereas this architecture was not initially defined with adaptation in mind?* A second question is related to safety issues due to dynamic updates: *any change at the functional level may have an impact on safety mechanisms, so how to adjust/improve safety mechanisms?* Safety mechanisms are, in particular, linked with Automotive Safety Integrity Level (ASIL) as defined in ISO 26262 [2]. We show that it is possible, within an automotive embedded system, to have dynamically updatable safety mechanisms that come with functional modifications.

Over-The-Air (OTA) updates aim at reducing the amount of data that need to transit to the vehicle and decreases significantly its downtime. In practice, such updates can only be done when the car is parked and powered-on, the driver being informed that software need to be updated. Such facilities could also be used during some specific phases when driving (e.g. platooning on highways) and related to car-to-X interactions, but this is out of the scope of this work today.

This paper is organized as follows. In Section II, we shortly describe the problem and explain our contributions. After a short overview of the context of the work and of relevant concepts of AUTOSAR we describe the basic principles regarding the introduction of flexibility within AUTOSAR in Section III. In Section IV, we discuss the whole process of updates development and integration. Section V focuses on updates of related safety mechanisms. Finally, we present the application of the various concepts developed in a case study in section VI and related work in Section VII. We finally draw the lessons learnt and conclude.

II. Problem Statement and Contributions

Updating partially an embedded software means that the new configuration is developed and validated off-line as a whole. This includes safety analysis and thus the determination of the impact of updates on system safety in general, and safety mechanisms in particular.

Our primary goal of adding functional updates in an AUTOSAR system must comply with the requirements of ISO 26262 [2]. This means that if we want our system to remain ISO-compliant, we must provide for ways to also update safety mechanisms. These safety mechanisms are twofold: some that are only related to the update added to the system, and some are more global and impact already existing safety mechanisms. An example of individual safety mechanisms is a wrapper around an update to control the validity of the input and output data. Existing safety mechanisms could also be modified if the update is integrated into a processing chain that has end-to-end safety mechanisms. In this case the existing mechanism could be modified if the constraints change when the chain is modified.

Fig. 1 presents the three possible types of allowed updates. It is possible to add i) a functional-only update, or ii) a

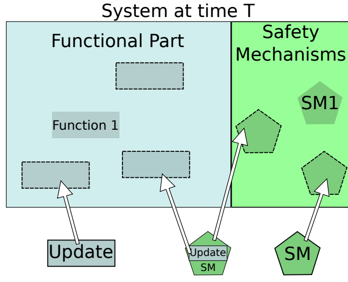

Fig. 1. Three examples of evolution of an AUTOSAR-based system

functional update with one or several associated safety mechanisms, or iii) an additional safety mechanism. To this end, we introduce few concepts in the application design beforehand, both for functional and non-functional updates that can be done remotely. We finally discuss the limits of the exercise and briefly draw the lessons learnt from this work.

III. Concepts for Updates

This section deals with the basic concepts for introducing flexibility within AUTOSAR systems. We start with defining what an update represents in AUTOSAR. We then present the notion of *Container* that is a cornerstone in the preparation of the application for receiving future updates. We also introduce the concept of *Smart Pointers* that corresponds to certified indirections that enable us to modify the application at runtime. Finally we describe our memory management mechanisms, that allow modifications of stored executables while maintaining application consistency.

A. AUTOSAR

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

1) Software Components (SWC): Software Components (SWCs) correspond to application functions. An SWC is defined as a group of fragments of executable code called *runnables* (e.g. a C function). A runnable implements a specific feature and can be executed either periodically or on the occurrence of an event (e.g. the reception of input data). In order to communicate with other SWCs or with the basic software layer, an SWC is associated with input and output ports.

2) Run-Time Environment (RTE): RTE can be seen as a collection of communication channels between different SWCs or between SWCs and the basic software. A communication channel enables data to be sent (resp. received) by an SWC to (resp. from) another SWC or an element of the basic software. Its role also includes triggering the execution of functions by sending events.

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 with the AUTOSAR standard. It implements a fixed priority scheduler, interrupt handling, and protection against unintended uses of OS services.

There are two types of tasks in AUTOSAR OS: basic tasks and extended tasks, the main difference being that an extended task may wait for an event while a basic cannot. Thus, basic tasks get synchronized at the beginning and at the end of their execution only, while extended tasks can synchronize with other tasks using events.

B. Update Definition

Section III-A defined the *runnable* to be the actual realization of functionalities, and Software Components (SWC) to be a hierarchical composition of runnables. Hence, the granularity of update we consider within AUTOSAR is the runnable. If it is possible to add a runnable, then we can add any kind of functional updates. Adding all the runnables of a Software Component means adding the Software Component itself. There are three types of possible updates: addition, modification, deletion of a functionality or a safety mechanism.

The updates usually cover a processing chain, which is itself made of several runnables that exchange data. Modifying an existing functionality does not impact the structure of the processing chain (no new data exchange is added to the chain), only the content of one or several runnables is changed. Adding a new functionality, on the other hand, consists in: adding one or several new runnables, adding at least one new communication channel (with previously existing elements of the processing chain, or between added elements), and modifying existing runnables that communicate with new elements to take into account new data. Last but not least, the impact of an update on execution time must be carefully taken into account, as described in Section IV-C

C. Hypotheses for Updates

In order to precisely define the accessible perimeter for updates in the (rigid) context of AUTOSAR, we provide here a set of working hypotheses.

a) Applicative updates only: The first and foremost hypothesis in this work is to focus on applicative updates only. In other words, we want to add functionalities above the RTE (and underlying communications when necessary). Such an hypothesis means that there are no modifications of the *Basic Software*, which in turn has some consequences.

First, the Operating System used in the context of AUTOSAR is static and does not allow for modifications *a posteriori*. Thus, the envisioned updates are assumed to be integrated within one or several existing tasks — no new task can be created for an update.

A second direct consequence of having no modification of the *Basic Software* is that distributed updates are not possible. Indeed, they would require a modification of the messages transiting on low level bus (e.g. CAN), and communication stack (within the *Basic Software*). Thus, the update we consider are confined within a single ECU. In case of distributed

updates, global consistency must be ensured and updates must be transactional.

b) Operating System and Architectural restrictions: All the systems we deal with are single core processors. Indeed, multi core may introduce new problems, such as deadlocks, that are beyond the scope of this work. We do not consider also the addition of new physical sensors or actuators, existing hardware and drivers in the BSW of the initial configuration are considered for the development of updates. Regarding the scheduling scheme, we assume a Rate Monotonic one [3].

c) Periodic vs. sporadic: Finally, we mostly focus on the addition of periodic updates. The reason for this is that event-triggered runnables are usually placed in extended tasks [4] which means that they need an OS-event to occur for resuming their execution. Yet, adding event-triggered runnables would require to add new OS-events and therefore modify the Operating System which is part of the Basic Software. Thus, we consider that only periodic runnables will be added to the system. This is consistent with the composition of most automotive application, since most of the functionalities are periodic (more than 70% for Renault testbed).

D. Basic concepts for updates

The two basic concepts introduced for updates are *Containers* and *Smart Pointers*.

A *Container* is the implementation of an adaptation space within the application. It acts as a placeholder that can afterwards be filled with an update. In this section we explain what a container is and how it can be used for placing functional updates. In a latter section, we will also develop how containers can also be used for safety purposes.

A *Container* holds characteristics that are statically fixed:

- Activation mode (periodic or event-triggered)
- Period (when applicable)
- Priority (inherited from the task it belongs to)
- Trigger (alarm for periodic, event for sporadic)
- Status (empty, filled with an update)

Note that we do not consider here the timing aspects and the Worst-Case Execution Time (WCET) of containers. This is because when a container is empty, it consumes hardly any time, and this is negligible. Instead, we use sensitivity analysis [5] in order to take into account the variation regarding real-time analysis and use an opportunistic approach to make sure each update can fit in.

Runnable 1 (SW1)	Runnable 3 (SW1)	Runnable 5 (SW2)	Container

Task 1

Fig. 2. Example of a task with a container

Fig. 2 presents an example of a task in which we have added a container. Therefore, this task not only contains runnables (from various SWCs), but also a container designed for harboring future updates. A design time, an abitraty number of empty containers can be added to tasks. Thus the tasks will be equipped to face any futur update scenario.

In order to allow the system to be updated, the notion of *Smart pointers* adds an extra level of indirections to the system. Indeed, in each task, we add an indirection table that is used to call all runnables and empty containers. This table is populated by *Smart Pointers* that redirect to the proper function to call, but also include a description and several safety mechanisms.

Primarily, a *Smart Pointer* contains a reference to the runnable that will be called to realize a function. Additionally it contains a description of this runnable that characterizes the container, namely its period, priority and status. This means that we can also use exactly the same mechanisms of *Smart Pointers* for the containers. The difference between the containers and the runnables in the table are their status (filled for a runnable, empty for a containers) and the reference that points to an actual function for the runnables and an empty function for the containers. A *Smart Pointer* also contains the ID of the runnable, which is a unique identifier for each runnable. Note that two versions of a same runnable have different IDs. All containers can point to the same empty function as its only purpose is to give an initial value to all pointers. Fig. 3 represents an example of indirection table in a task. There may be several containers in each task since they do not consume any time and the actual addition of an update is subject to a prior scheduling analysis.

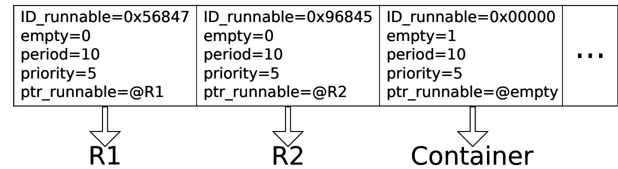

Fig. 3. Representation of indirection Table

IV. Off-line and On-line Process for Updates

In order to create and integrate an update, not only should the application be prepared beforehand with the container added to the architecture, but some steps are also necessary both off-line and on-line afterwards in order to create and integrate this update within the system.

Fig. 4 shows the various steps for creating and loading the application within the system. There are two steps for this, one which happens off-line for creating and validating the update and the second one that occurs on-line, within the ECU to integrate the update within the system.

In this section we detail three specific points related to this process: i) safety issues regarding smart pointers handling for modifying runnables and filling in containers, ii) memory management within the ECU and iii) scheduling verification. The first two points occur on-line for loading the update within the ECU while the scheduling analysis must occur off-line when the update is created and verified within its environment.

Note that when an update is created, it is paramount to make sure that all safety criteria are met. Therefore a careful safety analysis must be set up for determining all the implication of the update safety-wise, and consequently add or modify safety mechanisms, as described in Section V.

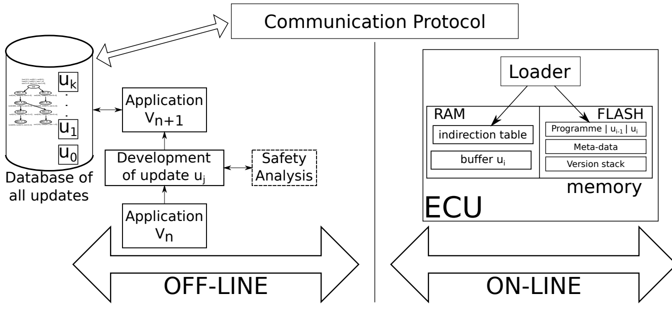

Fig. 4. On-line and off-line steps for creating and loading an update

A. Safety issues and Smart Pointers

Regarding safety, these *Smart Pointers* are a single point of failure. Indeed, if the value of a function gets corrupted, then a wrong segment of code could end up being executed, which could possibly have catastrophic consequences for the system. Smart Pointers must provide additional safety mechanisms.

The first safety mechanism of Smart Pointers is based on signatures, to check the value of the reference. Each time a function is called using a pointer, the value of this pointer is cross-checked with its signature. A consequence of this mechanism is a small overhead regarding execution time, but this overhead is negligible when compared to gains obtained from storing Smart Pointers in RAM.

By default, indirection tables for tasks are stored in RAM in order not to slow down the execution. Each time the ECU is turned off, the content of the RAM is erased. A backup of the *Smart Pointers* value is therefore written in FLASH. This backup is updated each time a modification occurs in the system (meaning that the update of the FLASH section that contains the backup of each indirection table is part of the global update process when a new functionality is added).

In a nutshell, the concept of *Smart Pointers* is the core building brick of each indirection table for OS tasks. To ensure safety of the references contained in these pointers, we add a signature of the reference, and backup indirection tables in non volatile memory. In the event that a discrepancy is found between a reference and its signature, a recovery mechanism loads non corrupted data stored in FLASH memory.

B. Memory Management

The way we have defined the various concepts does not take into account any reservation of memory space. Yet, it is important to have a sufficient amount of available non-volatile memory for storing updates. Moreover, there should be RAM memory available too for storing new variables. This section deals with the handling of those two types of memory.

Regarding non-volatile memory (automotive systems typically use FLASH memory), the problem that we face is that this kind of memory is sub-divided in sectors. Moreover, when erasing memory, the granularity of erasure is necessarily a complete sector, and in order to write, the memory must have been erased beforehand. Thus, one of our main hypothesis is that all of the updates will be written in FLASH memory after the current program. This means that we never erase previously loaded software. The reasons for this are twofold: first this enables to rollback in case of problem (since all o the previous versions of software are still available), second this means that we do not have the problem of erasing the memory (assuming that the initial system is loaded on a fully erased memory).

Another problem that we face is to determine at which memory addresses each update need to be written, in order to link properly these functions with the rest of the program. Namely, the *Smart Pointers* must have the proper reference for executing the corresponding function. For this reason, the memory addresses are handled off-line, meaning the ECU itself does not need to determine itself this part of the update process. This requires to know precisely for each version of the software the corresponding memory map.

C. Real-Time Impact of Updates

From a real-time perspective, the impact of an update can be twofold: (1) the modification of the WCET of some tasks; (2) the modification of the precedence constraints between tasks. These modifications can both have a negative effect on the schedulability of the system. Thus, it is paramount to make sure that each time an update is added to the system, the schedulability is unaffected. We describe how the modification of WCET and precedence constraints are dealt with independently and finally, we explain how all parameters can be taken into account in order to determine if an update can be added to a specific system.

1) Precedence Constraints: The concept of precedence, in the case of AUTOSAR application is related to data exchanges between tasks. A precedence constraint exists between task τ_i and task τ_j if τ_j cannot start its execution before τ_i completes its execution [6]. We consider that all data exchanges use the "last-is-best" scheme, meaning there is no buffering of data because the last data receive is the most relevant. Considering that the scheduling scheme is fixed priority, it is possible to use the priorities of tasks to resolve the precedence constraints: only tasks with a higher priority can send data to tasks with lower priorities.

In order to model precedence constraints between tasks, we use a precedence graph and a corresponding adjacency matrix. An example is given in Fig. 5. Without loss of generality, we assume that the tasks are ordered by decreasing priority : task τ_i has a higher priority than τ_{i+1}. Since tasks with higher priority may only send data to tasks with lower priority, if the adjacency matrix is lower and triangular then the precedence constraints are respected. We use this feature to check if an update is safe with respect to precedence constraints.

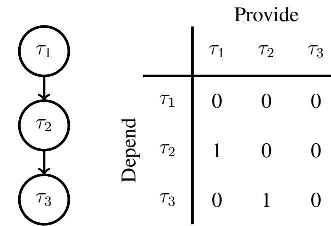

Fig. 5. Precedence graph (left) and adjacency matrix (right)

However, to use this feature some adaptation is required. In AUTOSAR data is exchanged between runnables, and therefore the precedence constraints are expressed for runnables

communications. The precedence constraint for the task set is inherited from the data exchange between runnables. A simple algorithm allows to determine the precedence matrix for the task set, given that data exchanges between runnables are known (from the design of the application), as well as the allocation of runnables to the tasks. This process is depicted in Algorithm 1.

```
Data: Runnables_Precedence_Matrix, Γ_N
Result: Tasks_Precedence_Matrix
∀ (α, β), Tasks_Precedence_Matrix(α, β) = 0 ;
for all (i, j) do
    if Runnables_Precedence_Matrix(i,j) = 1 then
        for all Tasks τ_k ∈ Γ_N do
            if Runnable_i ∈ τ_k then
                precede ← k
            end
            if Runnable_j ∈ τ_k then
                depend ← k
            end
        end
        Tasks_Precedence_Matrix(precede,depend) ← 1;
    end
end
```

Algorithm 1: Building the precedence matrix for set Γ_N

2) Sensitivity Analysis: To prove that installing an update is safe with respect to some real-time requirements, we need to know to what extend the WCET of the tasks can be increased while preserving the schedulability of the system. To do so, we use the *sensitivity analysis* proposed by Bini and Buttazo in [5].

In order to take advantage of this advanced result of real-time theory, we propose the following AUTOSAR task model:

- $\Gamma_N = \{\tau_1, \ldots, \tau_N\}$ the set of tasks,
- $\rho = \{R_1, \ldots, R_M\}$ the set of runnables,
- *Run*(τ_i) the set of all runnables embedded in task τ_i,
- C_{R_j} the WCET of runnable R_j,
- $C_i = \sum_{j \in Run(\tau_i)} C_{R_j}$ the WCET of task τ_i,
- T_i the period of task τ_i
- M$[\Gamma_N]$ the precedence matrix for tasks

In this model, task deadlines are considered to be equal to their periods. Moreover, in AUTOSAR a task can be configured with an activation offset. This allows to smooth the CPU workload and avoid traffic burst on the communication network. However, we do not take into account this parameter in the model. All task offsets are assumed to be equal to 0. It is a safe assumption: if the task set is schedulable with all offsets equal to 0 then it is schedulable with any choice of offsets [7].

The sensitivity analysis relies on the necessary and sufficient schedulability condition for fixed priority preemptive systems. It defines a region $\mathbb{M}_N$ which encompasses all the values of tasks WCET such that the system is schedulable. Adapting the result to our model, this region is given by:

$$\mathbb{M}_N = \{C_1, ..., C_N \in R_+^N :$$

$$\bigwedge_{i=1..N} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t \ \} \quad (1)$$

$$\text{with} \begin{cases} \mathcal{P}_0(t) = t \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \bigcup \mathcal{P}_{i-1}(t) \end{cases}$$

Using this region, it is possible to define metrics to evaluate the flexibility of the system. In particular, we can determine ΔC_k^{max}, the maximum increase of WCET a task τ_k can endure:

$$\Delta C_k^{max} = \min_{i=k,..,N} \max_{t \in \mathcal{P}_{i-1}(T_i)} \frac{t - \mathbf{n_i} \cdot \mathbf{C_i}}{\lceil t/T_k \rceil} \quad (2)$$

where $\mathbf{n_i}$ and $\mathbf{C_i}$ are vectors. $\mathbf{C_i}$ is the column vector of all execution times of tasks between task τ_1 and task τ_i and $\mathbf{n_i} = \left(\left\lceil \frac{t}{T_1} \right\rceil, \left\lceil \frac{t}{T_2} \right\rceil, ..., \left\lceil \frac{t}{T_{i-1}} \right\rceil, 1\right)$.

With such metrics, it is possible to verify if there is enough space in the system to add an update.

3) Update and Sensitivity Analysis: There are three necessary steps to add a new runnable within an existing system: determining containers compatible with the new runnable characteristics, selection of containers compatible with the precedence matrix, and proving the safety of the update with respect to sensitivity analysis.

First, all containers compatible with the characteristics of the runnable are determined. Once all possible locations have been found, the second step consists in determining the precedence matrix for the task set with this new runnable. The locations that prevent the matrix from remaining lower and triangular must be eliminated. The condition for promoting a location rather than another is to minimise the factor S defined as:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{i} M[\Gamma_N](i,j)$$

Finally, the last step is to use sensitivity analysis on our model of the task set. We use the approach proposed by Bini and Buttazzo[5]. The goal here is to prove that adding the update is safe with respect to schedulability and to keep the maximum global flexibility in the system. If there are still several locations that are acceptable, the chosen location is the one that maximises global flexibility (namely the ΔC_k^{max}).

V. Update of Safety Mechanisms

This section presents the safety mechanisms that should be added in an automotive system to ensure its proper function. It first focusses on ISO 26262 [2] mechanisms and Safety Integrity Level (ASIL), and details the extra mechanisms that we have added specifically for providing safe updates and be able to react appropriately in case of a problem. Then we detail how safety mechanisms should evolve with functionality updates. There are two levels of possible evolution for safety mechanisms: i) a fine grain level that enables verification at

TABLE I. SAFETY MECHANISMS FOR ERROR DETECTION AND HANDLING IN ISO 26262 FOR SOFTWARE

Criticality	Weak (ASIL A/B)	Medium (ASIL B/C)	High (ASIL C/D)
Data Flow	Range check on input and output, plausibility check, data error detection (data signature or redundancy)	Range check on input and output, plausibility check, data error detection (data signature or redundancy)	Range check on input and output, plausibility check, data error detection (data signature or redundancy)
Control Flow		Watchdog, Control Flow Monitoring	Watchdog, Control Flow Monitoring
Architecture	Static Recovery mechanisms	Static Recovery mechanisms	Static Recovery mechanisms, Diverse software design, Independent parallel redundancy

the runnable granularity, a feature that is not directly supported by AUTOSAR, and ii) a coarse-grain level allows the addition of end-to-end safety-related mechanisms.

A. Safety Mechanisms

1) ISO 26262 mechanisms: ISO 26262 [2] is the reference for safety in the automotive domain. It defines methods that should be used in order to obtain a dependable system. Although ISO is defined for the various steps of the development process of an E/E (Electrical and Electronic) system, we only focus on software safety in this work. The Automotive Safety Integrity Levels (ASILs) correspond to the criticality levels of the various functionalities of the system. ASILs range from A, which corresponds to less critical functions, to D that represents the highest criticality level.

There are three main points that require checking, namely data flow, control flow and architecture. Table I presents a summary of the safety mechanisms in correlation with ASIL and the various checking points.

When the ASIL increases, the number of verification and safety mechanisms increases. Yet, whatever the level of criticality, data flows need to be monitored to make sure that the problems that may arise in the system are not related to data transmission. For this reason, range check have to be performed on input and output data, as well as plausibility check (e.g. if the outside temperature is measured in summer, it is not likely that $-20°$C would be a valid value).

Control flow verification are only required for higher ASIL. In this case it is recommended to set up watchdog and control flow monitoring.

Finally, mechanisms are added at the architectural level to prevent the system downtimes. The basic mechanism that must be set up regardless of the ASIL is static recovery mechanisms, e.g. action of reset when an error is detected. Yet, when the functionality is truly critical (with very high ASIL), other mechanisms must also be added, such as diverse software design and independent parallel redundancy.

Finally during the design of the application, the principle of *Freedom From Interference* should be applied. This means, in particular, that a function with low level criticality must not interfere and cause errors within higher criticality functions.

2) Roll-Back: The current state of an ECU regarding the software version is handled by the ECU itself. This means that specific mechanisms must be added for being able to detect a problem and revert to a previous version of the software that was properly working. Moreover the ECU must be capable to handle this roll-back itself without any external instructions. For this purpose, we do not erase any of the previous versions within the ECU and implement a version stack that is stored in non-volatile memory. This stack enables to trace back all the versions that have been stored in the ECU.

This mechanism enables us to ensure dependability for the software by reversing to a safe version. Indeed, if a problem is detected when an error occurs, it is always possible to revert to a stable and working previous version that can be considered as degraded mode of operation.

B. Fine-Grain Safety Mechanisms

An interesting property of our approach is the possibility to handle safety mechanisms at a runnable level. This is not necessarily possible within the AUTOSAR standard. Indeed, runnables are allocated to tasks, and the granularity for verification is usually the tasks rather than the runnable itself. The only verifications that can be done are for the data exchanges by intercepting RTE data. This has been done for example by using safety wrappers [8]. However, this approach is static, as wrappers are inserted within the system before being loaded on the ECU and cannot be modified afterwards.

We offer the possibility to introduce mechanisms that can check data flow and control flow without modifying the structure of an AUTOSAR system. Moreover, we make it possible to implement safety verification at a very fine-grain level, namely at a runnable level. This means that verifications can be made before and after the execution of each runnable, if necessary. This is truly innovative given the way an AUTOSAR system is built. Indeed, the typical granularity level for safety verification is at task level since runnables are grouped into tasks. This means that if verifications are necessary at a runnable level, the safety mechanisms should be integrated directly within the runnable. Yet, separation of concerns between functional and non-functional parts is not achieved. Separation of concern is a key concept for maintenance and software evolution, as it has been demonstrated in many work, and that we consider in this work.

The idea for allowing fine grain safety mechanisms is to use the containers added in tasks in order to execute safety assertions rather than actual functional runnables. Moreover, our definitions of containers and smart pointers allow a lot a flexibility since it is possible with these mechanisms to modify the order of execution of the runnables and containers within a task. This means that it is possible to modify the code executed within a task so that safety mechanisms execute before and after each runnable. This only requires a sufficient number of containers and a real-time verification. The real-time model is then slightly modified: the WCET of a task is the sum of all WCET of its functional parts (runnables) and non-functional safety mechanisms execution time.

Fig. 6 shows how the fine-grain updates of safety mechanisms can be implemented. These are mechanisms that need to be placed directly within the task around the runnable

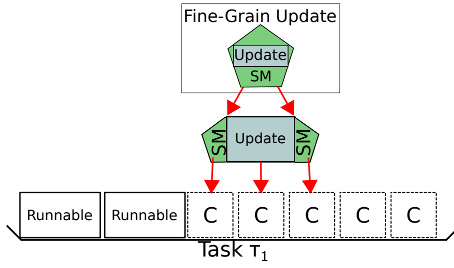

Fig. 6. Example of fine-grain update loaded within a task of the system

whose behavior they are verifying. These safety mechanisms act as a wrapper around the runnable, e.g. to verify input and output data. Fig. 6 shows an example where such safety mechanisms surrounding the update are placed in containers directly adjacent to the one where the update is placed. This allows to perform verifications immediately before and immediately after the execution of a runnable.

C. Coarse-Grain Safety Mechanisms

Other important mechanisms for safety are coarse-grain mechanisms that for example allow a verification of end-to-end timing constraints. These mechanisms need to be executed at a task level rather than at a runnable level. In this section we describe how to execute specific code before and after task execution, and to be able to modify what to execute.

AUTOSAR OS provides for *Pre-Task Hooks* and *Post-Task Hooks* [9], which are specific routines that execute before (*Pre-Task Hook*) or after (*Post-Task Hook*) a given task. For adding new mechanisms or modify existing mechanisms placed into these Hooks, we use the same concepts and mechanisms, namely *Containers* with *Smart Pointers*.

Thus in each hook, we add an indirection table and non functional containers for executing existing safety mechanisms, but also modify them and add new ones. The specific way of adding this indirection level and the containers is strictly identical to what is done for functional parts. However, only non-functional mechanisms can be placed within pre- and post-task hooks. Moreover, regarding real-time analysis, the WCET of each safety mechanisms added within the hooks must be added to the WCET of the tasks to make sure that the system remains schedulable.

This kind of safety monitors is typically for inter-function verifications and covers the execution of several runnables. When updating these, there are two possibilities: either for runnables that all belong to the same task or for runnables that belong to different tasks. In the first case, the safety monitors are added in the pre- and post task hooks of the same task, and this corresponds to intra-task mechanisms. Otherwise, the pre-task hook of one task (e.g. τ_2 in Fig. 7) and the post task hook of another task (e.g. τ_1 in Fig. 7) are modified and this corresponds to inter-tasks mechanisms.

Coarse grain safety mechanisms are used for end-to-end verification of timing constraints, but also for checking the consistency of the results produced for/by runnables belonging to the same processing chain.

In a nutshell, our container mechanism can be applied both to functional (tasks) and non-functional (hooks) parts, enabling

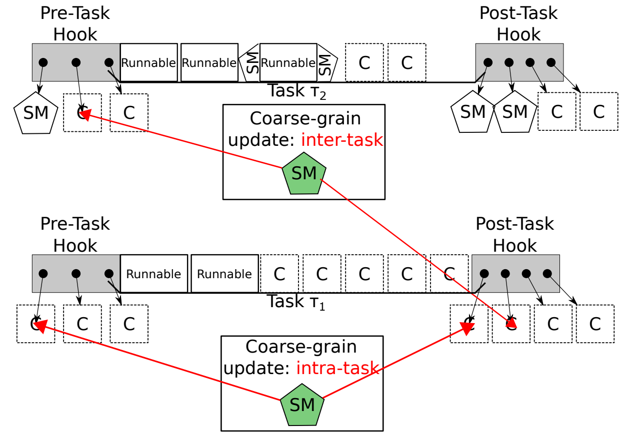

Fig. 7. Two types of coarse grain safety mechanisms

the update of safety mechanisms when a functional update occurs. It is therefore possible to add new safety mechanisms attached to a specific update, and to modify existing ones.

VI. Case Study

All the concepts for functional updates and safety management can be applied on an example application that handles the blinkers in a vehicle. This application provided by Renault Engineering was slightly modified for this work. In this section, we show how all the concepts are implemented. Moreover, we intend to demonstrate here, that even for an application that can seem fairly trivial as the blinker, there is actually a specific regulation that must be followed and some safety goals are derived from this.

A. European Regulation for Blinkers

To begin with, it is important to highlight that there is a specific European regulation for the blinker [10], not only regarding the placement of the blinkers on a vehicle, but also for its behavior.

The regulation regarding behavior states that all blinkers on the same side of a vehicle should be synchronized and triggered by the same physical controller. They should blink 90 ± 30 times per minute and at most one second should elapse between the first activation of sensor and the effect on the actuator (first time the light bulb is switched on). Then, at most half a second later the indicator should turn off for the first time.

B. Description of Blinker Application

1) Development Process and Preparation of the Application for Updates: For developing the blinker application we went through a number of steps going from the functional needs (partly dictated by European regulation presented above), to the final implementation. Fig. 9 shows the various steps that need to be gone through for developing any AUTOSAR application. Note that the steps that are shown here are only for software development.

The standard development process is slightly modified for introducing the necessary concepts and mechanisms for updates. Yet, it remains fully compliant with the AUTOSAR

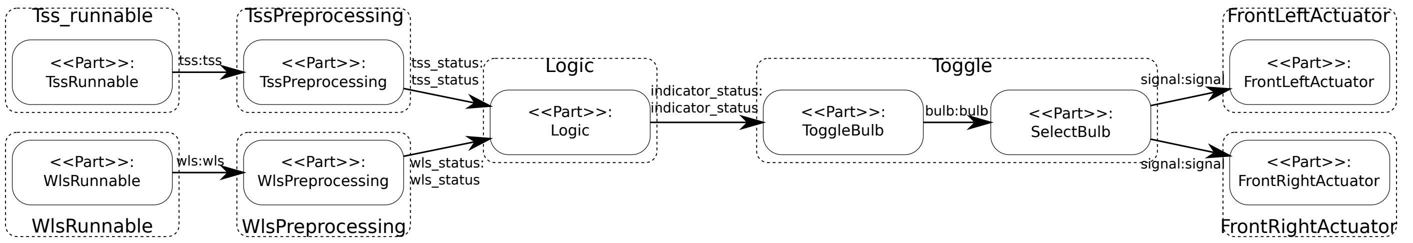

Fig. 8. Final Design for Blinker Application

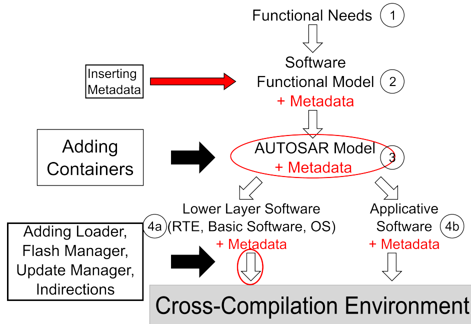

Fig. 9. Modified AUTOSAR development process

methodology [11]. There are three elements that are introduced within this process for allowing future updates. First we inserted meta-data that act as a form of signature of the application in order to trace back all elements inserted within the system. We also introduce both the non-functional and the functional containers within the tasks and the pre- and post-task hooks. Finally we add some low level mechanisms that aim at actually loading the update within the ECU.

2) Software Functional Model: This step is necessary for determining the various elements required in the application. It is not part of the AUTOSAR methodology and we have specifically added it for improving the design of automotive embedded application. It corresponds to step 2 in Fig. 9 and aims at designing the applicative layer of the software. In our approach, this model is made of a set UML (Unified Modeling Language) diagrams and constraints. We start with a *use case diagram* that presents the possible use-cases of the function we are designing. In our example, there are two use cases, namely the turn indicator and the warn lights.

The next step of this modeling consists in designing separately each use case and identifying every element that is necessary to fulfill it and how these elements interact. For this, we use for example sequence and activity diagrams to model data flow, control flow and timing properties.

Then, once this is done for each use case, we move to an advanced design where the elements that are common in each use case need to be merged. In our blinker application we merge the parts that treats data and send status to light bulb. The chain that processes specifically data from turn indicator and warn lights remain separated. The final product is a complete processing chain that realizes both the turn indicator and the warn lights. Finally, the objective is to obtain runnables by grouping elements within the processing chain that are functionally complementary.

Fig. 8 presents the final architecture of the blinker. Dotted rectangles correspond to the actual runnables that will then be used in our AUTOSAR model while rectangles with rounded corners are the elementary actions that are manipulated throughout the design in UML. Our final application is made of 8 runnables (AUTOSAR Model).

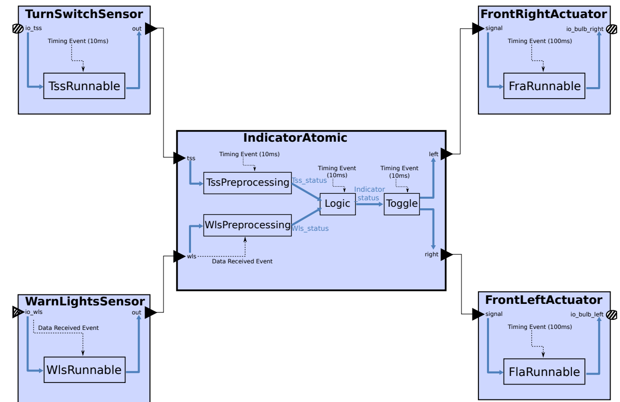

Fig. 10. AUTOSAR model of Blinker Application

3) AUTOSAR Model: From the runnables that are identified in the high level software model, the next step is to have the complete AUTOSAR model. Note that here we are only interested in the applicative software and therefore we only show the allocation of runnable to software components, the communications between these elements and the timing properties. This last point is crucial for later allocating the runnables to the OS tasks.

Fig. 10 shows the AUTOSAR model composed of 8 runnables. They are split amongst 5 SWCs: one for each sensor or actuator, and one for data processing and status sending to actuators.

C. Safety Analysis

In this section we show a FMECA (Failure Modes, Effects and Criticality Analysis) [12] table in order to provide a complete safety analysis for our blinker function. We only present an excerpt from the complete FMECA table for simplification purposes and due to space limitation.

The system we study handles both the turn indicator part and the warn light part. The appropriate light bulb(s) (actuators) must blink in accordance with the sensor that is activated. This system presents two safety goals:

- The wrong blinker should not be activated (ASIL B)
- Blinkers shall comply with timing constraints from european regulation [10] (ASIL A).

TABLE II. FMECA TABLE FOR THE BLINKER APPLICATION (EXCERPT)

Item	FM	Potential Causes	Local Effects	Upper-Level Effects	Safety Level	SSM	ULE with SSM
1.1	FM1	Sensor Problem or No data read/written in RTE channel	No data transmitted to blinker processing element	No blinker is activated when sensor is pressed	QM	-	-
	FM2	Wrong data read/written in RTE	Wrong data is transmitted and processed	Wrong blinker is activated	ASIL B	Data redundancy, data signature	no effect
3.1	FM1	Scheduling Problem	Data is transmitted too late, end-to-end timing violation	Blinker not compliant with European regulation, user may consider blinker bulb faulty	ASIL A	End-to-End Verification	no effect
	FM2	No data read/written in the RTE	No data is transmitted to the actuator	Blinker fails to activate	QM	-	-
	FM3	Wrong data read/written in the RTE	Wrong data sent to actuators	Blinker activates when it should not or fails to activate. Turn indicator is activated instead of warn light (loss of consistency)	ASIL B	Data redundancy, Data signature	No effect

There is also one undesirable event that is only graded with Quality Management (QM) in ISO 26262: a problem occurs when a sensor is activated and nothing happens on the blinker.

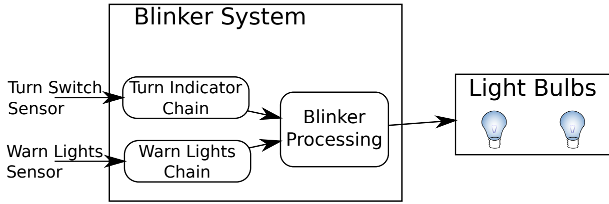

Fig. 11. High level schematic of blinker system

Fig. 11 presents a high level view of the software part of the blinker system. Our application is divided into two main processing chains: one that handles the turn indicator and the other that handles the warn lights. Then there is a part that is common for the turn indicator and the warn light, which is called here "Blinker Processing". This last element then communicates with the actuator, namely the left and the right light bulb.

Product	Functional Requirements
Turn Indicator Chain	1.1 Send turn indicator status
Warn Lights Chain	2.1 Send warn lights status
Blinker Processing	3.1 Send right blinker signal 3.2 Send left blinker signal

TABLE III. FUNCTIONAL REQUIREMENTS OF THE PRODUCTS

The exact description of the various blocks presented in Fig. 11 is shown in Table III. This table gives the functional requirements of each of the elements in the blinker system. It is important to note that these elements correspond to software parts. We do not detail here the underlying hardware.

Functional Reqt.	#	Failure Mode	Product Effect
1.1 Send turn indicator status	FM1	Loss of turn indicator status in chain	Blinker fails to activate
	FM2	Erroneous turn indicator status	Wrong indicator is activated
3.1 Send right blinker signal	FM1	Wrong timing for sending data (end-to-end timing constraint violated)	Blinker no longer compliant with European regulation, blinkers do not activate on time
	FM2	Loss of right blinker signal	Right blinker fails to activate
	FM3	Erroneous value send to right blinker	Right blinker activated instead of left, right blinker fails to activate

TABLE IV. FAILURE MODE ANALYSIS OF THE SYSTEM (EXCERPT)

To keep the safety analysis as concise as possible, we only present here the various steps for the turn indicator chain and for the right light bulb. The analysis for the warn light chain is fairly similar to the analysis for the turn indicator chain and the right light bulb is completely symmetrical to the left one.

Task	Run_i	C_i
$Task_Wls$	WlsRun (2.45 ms)	2.45
$Task_WlsP$	WlsP (0.15 ms)	0.15 ms
$Task_10ms$	TssRun (2.8 ms), TssP (0.075 ms), Logic (1.075 ms), Toggle (1.425 ms)	5.375 ms
$Task_100ms$	FlaR (0.675 ms), FraR (0.675 ms)	1.35 ms

TABLE V. CHARACTERISTICS OF TASKS (BLINKER APPLICATION)

After determining the various products, the second step is to determine what are the possible failure modes for our functional requirements. An excerpt of the failure mode analysis and the product effect is presented in Table IV. We can see here that for turn indicator there are two failure modes that are related to the data exchange, while for the blinker processing there are not only failure mode related to problem occurring in the data sending but also regarding timing constraints and European regulation.

This analysis is then further developed with the actual FMECA presented in Table II. Note that this is only the part of the FMECA that corresponds to the failure modes presented in table IV. The fault model we consider in this analysis is related to physical faults. This table shows how we can use the safety mechanisms recommended by ISO 26262 can be placed within a given system in order to mitigate the effect of faults that can occur within this system.

D. Real-Time Analysis

After the runnables are allocated to the tasks of the OS, We have 2 periodic tasks and 4 event-triggered tasks. In this section we give a description of our system following the model given by section IV-C2.

$$\Gamma_N = \{Task_Wls, Task_WlsP, Task_10ms, Task_100ms\}$$

The WCET for the runnables presented in Table V are obtained using the toolbox Otawa [13], with a very simple representation of the microcontroller architecture. We use PowerPC hardware in the familly of MPC5510 controllers.

As already mentioned, the execution time of the safety mechanisms that must be placed within containers (for fine grain safety), or within pre- and post-tasks hooks (for coarse grain safety) also need to be added to the WCET of the task. These safety mechanisms for the present application are mostly data verifications (placed before and after runnables), and end-to-end verifications for timing constraints. The latter

TABLE VI. FMECA FOR THE PULSED TURN INDICATOR

Item	FM	Potential Causes	Local Effects	Upper-Level Effects	Safety Level	SSM	ULE with SSM
4.1	FM1	Sensor Problem or No data read/written in RTE channel	No data transmitted to blinker processing element	Blinker does not activate 3 times	QM	-	-
	FM2	Wrong data read/written in RTE	Wrong data is transmitted and processed	Wrong blinker is activated, blinker activated during an incorrect time	ASIL B	End-to-End verification, Data redundancy & signature	no effect

corresponds to inter-tasks verifications between the beginning of the turn indicator processing chain and the end of this chain (ASIL B constraint). For this reason, the first part is placed before the $Task_10ms$ and the end part after $Task_100ms$. This increases the execution time of each task using this mechanism by 0.20 ms.

There are also data verification mechanisms that must be placed for all elements of the turn indicator processing chain, namely every runnable except from WlsRun and WlsP. We estimate that this increases the execution time of each runnable of 0.15 ms for verifying data before and after its execution.

And finally the precedence matrix for our tasks, obtained using Algorithm 1, is shown on Fig. 12, where the tasks are sorted by priority: $Task_Wls$ is the task with the highest priority and $Task_100ms$ with the lowest priority. In this context we can clearly see that the matrix is lower and triangular. Note that the "1" that appears on the diagonal only means that there are two runnable within the same task that exchange data. All the precedence constraints of the system can therefore be fulfilled.

$\Gamma_N =$

	Task_100ms	Task_10ms	Task_WlsP	Task_Wls
Task_Wls	0	0	0	0
Task_WlsP	1	0	0	0
Task_10ms	0	1	1	0
Task_100ms	0	0	1	0

Fig. 12. Precedence matrix of tasks (blinker application)

E. Example of Update

In this case study we focus on one specific upgrade of the system, which is the *pulsed turn indicator*. This section details the characteristics of this update, the impact of this update on both safety and real-time properties, and shows how these issues are handled by the mechanisms we set up.

1) Design of the update: To design the *Pulsed Turn Indicator* update, we follow the same steps as for designing any use case. The objective of this update is to have a turn indicator blink 3 times in case of a short impulse on the corresponding sensor. This feature can be used in the case of lane change.

We start with an individual design of this function, with the corresponding processing chain, activity diagram, sequence diagram and timing and safety properties. Based on this design, we then evaluate the impact of this new functionality on the initial processing chain (as presented by Fig. 8).

We need an element able to detect the impulse on the sensor and able to adjust its output so that the corresponding actuator blinks 3 times. In our design, this role is held by the element "TssPreprocessing" on Fig. 8. The impact of the update will be the modification of the functional behavior of this element.

Fig. 13 shows the difference between the standard behavior and the expected behavior of the turn indicator when the pulsed option is added to the system.

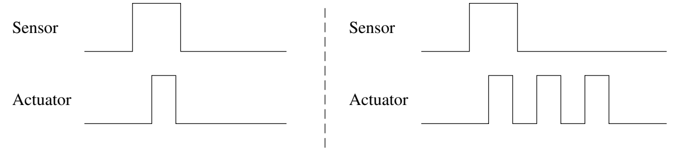

Fig. 13. behavior of regular turn indicator (left) and pulsed turn indicator (right)

2) Safety Analysis: Although the update we consider is more a modification of the system rather than an addition of new elements, for the safety analysis, we consider it as a new part to better study the impacts. Thus, Fig. 14 shows how the pulsed turn indicator interacts with the rest of the system already present. Tables II, III and IV need to be revisited to study the impact of updates within the system and determine which safety mechanisms need to be added or modified.

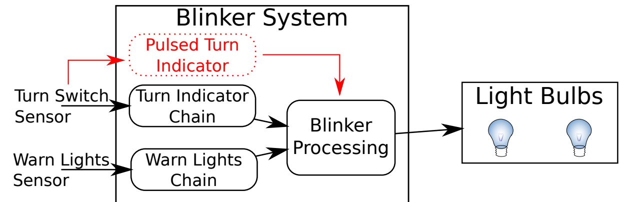

Fig. 14. High level schematic of blinker system with update

There is only one functional requirement that is attached to the *Pulsed Turn Indicator Chain* which is "Send pulsed turn indicator status". Based on this new functional requirement we can determine the failure modes and the product effects of the update. This analysis is presented in Table VII.

Finally, Table VI shows the impact of the update on safety mechanisms and how to mitigate the effects. The update has an impact on an ASIL B function. Therefore, safety mechanisms must be added in order to make sure that the data exchanged between the parts of the system is not corrupted. For example we can add cryptographic signature to check correctness of each data item in the processing chain.

Functional Requirement	#	Failure Mode	Product Effect
4.1 Send pulsed turn indicator status	FM1	Loss of turn indicator status in chain	Blinker does not blink three times. Standard behavior is observed
	FM2	Erroneous turn indicator status	Wrong indicator is activated, Blinker is triggered 3 times when it should not be activated

TABLE VII. FAILURE MODE ANALYSIS FOR THE UPDATE

Regarding the timing properties, some end-to-end verification need to be added to make sure that pulsed turn indicator is properly triggered and blinks 3 times. This means that the initial safety mechanism, that is only doing timing constraints verification, now needs to verify that the proper functionality is triggered by checking the number of activations.

The verification for data correctness need to be placed within containers (placed directly within the body of the task) and executed directly before the execution of the runnable, and after it completes its execution to verify consistency. We also need to modify end-to-end verifications that are placed within pre- and post-tasks hooks. The verification in this case is inter-tasks verification since we need to cross-check the consistency between the input and the output of the chain.

3) Real-Time Analysis: The impact of pulsed turn indicator on the system consists in:

- modifying the "TssPreprocessing" runnable. The only modification is the increase of the WCET of this runnable is now 0.5 ms. It remains in the same task.
- adding data verifications before and after the execution of the runnable within containers. These verifications are similar to what was described previously, and add 0.15ms to the execution time of the task.
- modifying the end part of the end-to-end verification that is placed within the post-task hook of $Task_100ms$. This mechanism is significantly more evolved that the initial mechanism, therefore its execution time grows. We estimate this increase in execution time to be about 0.1 ms.

This means that the new WCET for $Task_10ms$ is now 6,75 (sum of all WCET of runnables + all WCET of safety mechanisms for runnables and pre- and post- task hooks). $Task_100ms$ has a new WCET of 1,95. We use sensitivity analysis to check that the system remains within the boundaries of the scheduling space, and we find that despite having an utilization factor that is above 95%, the system remains schedulable.

Note that we do not edit the precedence matrix since in our example no new communication need to be added. Therefore, the precedence matrix remains identical.

VII. Related Work

A. Updates in Real-Time Automotive Embedded Systems

For introducing more flexibility in automotive embedded systems, several approaches are possibles depending on the level of these updates. We distinguish two main levels, which are model-based updates and binary-based updates.

For model-based updates, Becker *et al.* offer a tool that allows to define possible reconfigurations for a given system, regarding the context of execution. This tool requires a model of all these reconfigurations, and is easily interfaced with AUTOSAR systems. Yet its major drawback is that it is only compatible with a specific industrial tool called system Desk. A more general approach [14] proposes a tool chain based on UML an is designed for general safety critical real-time systems. It is quite an interesting approach, but it is not specifically design for automotive systems, let alone AUTOSAR systems.

Binary updates for embedded systems correspond to a necessary tool, as they allow for modifying part of the memory of the system without re-loading the complete software. This mean that updates are based on a system that build a tree of differences between two versions of binary code. For example bdiff [15] or vdelta [15] are implementations of such systems. In the domain of automotive software, Nakanishi *et al.* offer a method for creating low-level binary patched for updates. This can be highly interesting for updates OTA (Over-The-Air) since it reduces the amount of software that need to be transferred. Yet, these are merely tools, but a high-level analysis of the updates is required beforehand.

B. Safety in Automotive Systems

Nowadays, there is an increasing amount of functionalities that are handled with embedded software. Granted that the number of ECUs in a vehicle is limited (for reasons of cost), this means that unrelated functionalities can be allocated to a same ECU which could lead to interference problems [16]. Software safety, as defined in [17] is therefore a relevant matter and new concepts have been developed to improve safety for automotive software.

Most of the current work regarding safety in automotive embedded systems relies on mechanisms that are added a priori. Although some of these mechanisms can provide some flexibility at run-time, to the best of our knowledge we are not aware of actual systems that allow to add new safety mechanisms at run time as in our approach.

For example, Heckemann *et al.* develop the concept of *Safety Cage* [18] that allows for a formal verification of the system behavior depending on the context of execution. Another solution consists in introducing mechanisms in the system for self-reconfiguration [19] in order to prevent downtime. Another very wide spread method consists in adding online verification mechanisms beforehand in the system. For example, an approach consists in adding a *Safety Bag* which is a set of assertions that analyses data an can correct the effects on actuators [20]. It is also possible to add run-time monitors that check the proper execution of the code [21].

In AUTOSAR application, it is possible to add wrappers in order to instrument the application [8]. This enables the users to access produced data and modify them for fault injection, and to test the capacity of fault tolerance for the system.

All these approaches have this in common that they rely on safety mechanisms that are introduced beforehand in the system and that cannot be modified afterwards at run time. What we provide in this work is a way to have evolving safety mechanisms in the system. This property can be used either in correlation with evolution in the functional part of the application, or only for evolutions of the safety mechanisms.

VIII. Lessons learnt and conclusion

Adaptation of embedded systems was not a main concern in many critical application domains up to now. The economic pressure, for time-to-market reasons, but also the new challenges posed by the Internet of Things, impose a rapid

evolution of embedded systems. Smart cities are examples of a specific Internet of Things where cars become connected objects. These new trends combined with the recent innovation concerning autonomous driving and ADAS (Advanced Driver Assistance Systems) raise the problem of seamless evolution of critical systems. Both functional and non-functional safety issues have to be tackled as soon as critical embedded application are concerned. As mentioned in the Tesla Motors website[1], for critical functions like the autopilot: "*Autopilot features are progressively enabled over time with software updates*". An important point to mention here is that with the same set of sensors (cameras, radars, sonars) and actuators (command and control systems on engine, brakes, steering, etc. ...), a lot of improvement and new functions can be added over-the-air. This trend explains why software-intensive systems are being developed very fast and their complexity is increasing very much, leading thus to more evolutions that should be sent quickly using remote software maintenance.

This was one of the main motivations behind this paper: *will AUTOSAR be able to support these evolutions?* Our work shows that the AUTOSAR architecture as it is today does not offer enough flexibility to perform remote partial updates at this envisioned scale. This is due to its basic concepts, the software technology used, the tool-assisted development process that, although it provides production facilities, brings many constraints.

Although the current AUTOSAR architecture does not comply with the new trends discussed above, we have shown that remote partial updates are however possible. We have also taken into account the specific questions raised by the updates regarding real-time issues and safety.

We propose in the paper specific concepts, *Smart Pointers* and *Containers*, to allow for both functional and non-functional updates to be placed within an AUTOSAR system after the ECU has been loaded. We have shown that these concepts can be introduced in the development process in a seamless fashion. We have listed the assumptions enabling to change only the relevant parts of the software rather than reloading everything. The concepts that we introduced can be further used to implement safety mechanisms at a very fine grain level and fulfilling the constraint of separation of concern. The latter is very relevant for both safety and maintenance.

We make it possible to have evolving safety mechanisms, not only on a fine grain level, but also for intra- and inter-tasks verifications. Finally, we have shown on the example of the blinker application how an update can be added to the system while still respecting the safety and real-time constraints.

The final conclusion is two-fold. On the positive side, partial updates are indeed possible in *Classic AUTOSAR*, with very few concepts that are compatible with the standard as it is today. However, the limits we observed can only be solved with a new architecture (more conventional OS, component-based technologies, etc. ...) and this is exactly the focus of the *Adaptive AUTOSAR* recent initiative.

REFERENCES

[1] AUTOSAR Development Cooperation, htt://www.autosar.org.

[2] ISO TC22/SC3/WG16, "Iso 26262 - road vehicles functional safety," november 2011.

[3] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings Real Time Systems Symposium*, Dec 1989, pp. 166–171.

[4] OSEK Group, "OSEK/VDX Operating System (Release 2.2.3)," 2005, http://portal.osek-vdx.org/.

[5] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *Real-Time Systems, 2006. 18th Euromicro Conference on*, 2006, pp. 10 pp.–22.

[6] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, April 2010, pp. 301–310.

[7] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[8] T. Piper, S. Winter, P. Manns, and N. Suri, "Instrumenting autosar for dependability assessment: A guidance framework," in *DSN*, 2012, pp. 1–12.

[9] AUTOSAR, "Specification of operating system (release 4.0)," 2011.

[10] UNECE, "E/ece/trans/505," http://www.unece.org/fileadmin/DAM/trans/main/wp29/wp29regs/r48r6e.pdf.

[11] AUTOSAR, "Methodology," March 2014. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-1/methodology-templates/methodology/auxiliary/AUTOSAR_TR_Methodology.pdf

[12] European Cooperation For Space Standardization, "Failure Modes, Effects, and Criticality Analysis (FMECA)," http://www.ecss.nl/forums/ecss/dispatch.cgi/home/showFile/100704/d20080806093054/No/ecss-q-30-02b-draft2%2830April2008%29.pdf, April 2008, eCSS-Q-30-02B Draft 2.

[13] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: an open toolbox for adaptive wcet analysis," in *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, October 2010.

[14] J. Shen, X. Sun, G. Huang, W. Jiao, Y. Sun, and H. Mei, "Towards a unified formal model for supporting mechanisms of dynamic component update," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 80–89, Sept. 2005.

[15] D. Salomon, *Data Compression: The Complete Reference*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, pp. 850–940.

[16] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 33–42.

[17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[18] K. Heckemann, M. Gesell, T. Pfister, K. Berns, K. Schneider, and M. Trapp, "Safe automotive software," in *Proceedings of the 15th international conference on Knowledge-based and intelligent information and engineering systems - Volume Part IV*, ser. KES'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 167–176.

[19] M. Trapp, R. Adler, M. Förster, and J. Junger, "Runtime adaptation in safety-critical automotive systems," in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, ser. SE'07. Anaheim, CA, USA: ACTA Press, 2007, pp. 308–315.

[20] 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.

[21] S. Cotard, S. Faucou, J.-L. Bechennec, A. Queudet, and Y. Trinquet, "A data flow monitoring service based on runtime verification for autosar," in *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS),*, 2012, pp. 1508–1515.

[1] www.teslamotors.com