Design and Implementation of a Dynamic Component Model for Federated AUTOSAR Systems

Ze Ni
Software and Systems
Engineering Laboratory
SICS Swedish ICT AB
zeni@sics.se

Avenir Kobetski
Software and Systems
Engineering Laboratory
SICS Swedish ICT AB
avenir@sics.se

Jakob Axelsson
Software and Systems
Engineering Laboratory
SICS Swedish ICT AB
jax@sics.se

ABSTRACT

The automotive industry has recently agreed upon the embedded software standard AUTOSAR, which structures an application into reusable components that can be deployed using a configuration scheme. However, this configuration takes place at design time, with no provision for dynamically installing components to reconfigure the system. In this paper, we present the design and implementation of a dynamic component model that extends AUTOSAR with the possibility to add plug-in components at runtime. This opens up for shorter deployment time for new functions; opportunities for vehicles to participate in federated embedded systems; and involvement of third-party software developers.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures; D.2.12 [**Inter-operability**]: Distributed objects

Keywords

AUTOSAR, Software Components, Dynamically Reconfigurable Software, Federated Embedded Systems

1. INTRODUCTION

Automotive embedded systems are expanding rapidly in functionality and complexity, and a car nowadays typically contains several dozen electronic control units (ECUs) that are connected through communication networks. Each ECU runs control functionality using sensors and actuators, but there are also control functions that are distributed over several ECUs. The vehicular industry commonly relies on external suppliers of both ECU hardware and software, and with rising complexity, it has become increasingly costly to integrate these ECUs into a functioning system.

To cope with the increasing complexity of in-vehicle software, the automotive industry has for the last decade been developing the standard Automotive Open System Architecture (AUTOSAR) [7]. It decouples the basic software that needs to exist in all ECUs, from the application software which is specific to that node. It also provides a component model that eases reuse of parts of the application software, and allows it to be redistributed if the underlying hardware topology is changed, thereby improving flexibility and scalability. The automotive industry is very cost sensitive, and ECU hardware is traditionally kept to a minimum. Therefore, AUTOSAR has been designed to execute with limited resources and hence configuration of the system is done at design time with no structural dynamics during execution.

By making it possible to dynamically reconfigure parts of an AUTOSAR system, several benefits could be reached. Firstly, it would drastically decrease the time to market since software can be added or modified very late in the development process, and even allow feature upgrades in already produced vehicles. Secondly, in combination with external wireless communication, it gives the possibility for creating federated embedded systems (FES) [9], i.e. embedded systems in different products that cooperate with each other. Thirdly, it would create a foundation for open innovation where an ecosystem of third party developers can develop new services that add to the value of the products.

The purpose of this paper is to describe the implementation driven design of a dynamic component model that allows installation of plug-in components on top of statically configured software. This involves extensions to the architectural concepts in embedded software, first outlined in [3], and off-board support systems that are paramount for the configuration and management of the dynamic components.

The paper is organized as follows. Section 2 gives an introduction to central AUTOSAR concepts. Section 3 contains the main contributions of this paper, namely the dynamic component model and its life cycle management. Section 4 demonstrates the concepts on an example, inspired by a test platform, implemented in this work. Section 5 reviews the related work, while Section 6 concludes the paper.

2. OVERVIEW OF AUTOSAR

AUTOSAR is a layered software architecture that decouples application software (ASW) from lower level basic software (BSW) by means of a standardized middleware called runtime environment (RTE).

The BSW consists of an operating system that has evolved from the OSEK standard [10]; system services for, e.g., memory management; communication concepts; ECU and microcontroller hardware abstractions; and complex device drivers for direct access to hardware. Since the underlying hardware is generally resource constrained, the BSW is typically limited to such functionality that can be statically pre-defined. An example of limitations is that neither file system management nor dynamic memory handling is typically supported.

The ASW consists of a number of software components (SW-C). Each SW-C declares a number of ports for communication with the rest of the system. They can be either required ports (meaning that the component is expecting some input) or provided ports (used by the SW-C for its output). The ports can implement different interaction schemes, including sender-receiver or client-server. The internal functionality, or the runnable, of the component only accesses its ports, and not any other components, which promotes reuse and transferability of SW-Cs between ECUs. The runnables are mapped to OS tasks. SW-Cs can also be composite, i.e. containing other SW-Cs inside.

The communication between SW-Cs, as well as between SW-Cs and the underlying software layers, is based on a concept called Virtual Function Bus (VFB). The idea of VFB is to allow SW-Cs to communicate with each other as if they were all allocated to the same ECU. If they are in fact on different ECUs in a particular implementation, the communication between them has to be mapped to network messages without involving the SW-Cs themselves.

RTE is the realisation of VFB, providing the actual message transfer to and from SW-C ports. RTE provides an API to the ASW, and in turn calls the API of the BSW. Apart from communication, the RTE also handles other functionality, such as events, critical sections, etc.

In addition to technical concepts, AUTOSAR provides a development methodology which relies on different tools for software configuration, including BSW composition, allocation of SW-Cs to ECUs, and dependencies between SW-C ports, which are defined statically at design time in a number of description files. These files are then processed by AUTOSAR tools, creating executable software that implement the BSW, RTE, and ASW for a particular ECU.

Although AUTOSAR provides a lot of flexibility in reconfiguring a system, it does not offer any possibility to make dynamic additions, but any changes require the software to be rebuilt and the ECU to be reprogrammed.

3. THE DYNAMIC AUTOSAR CONCEPT

In the following, the key contributions of this paper are presented, describing a dynamic software concept that extends the AUTOSAR architecture to allow software reconfiguration at runtime. The first part looks inwards, proposing additional structural components and relations for making the ECU software reconfigurable. The second part takes a higher view, discussing necessary support structures for managing the additional software throughout its life cycle.

3.1 Dynamic Component Model

Several considerations are important in the design of a dynamic AUTOSAR architecture. First of all, it should fit in with the current practice of automotive software, adapting to such standardized AUTOSAR concepts as SW-Cs, communication ports, RTE, etc. At the same time, it should be non-static, both regarding software management and communication channels. In parallel, it is utterly important that the additional software (further called plug-ins) does not compromise the built-in functionality. These issues are addressed by introducing special-purpose SW-Cs, a dedicated runtime environment for the plug-ins, and a number of special purpose SW-C ports, as presented below.

3.1.1 Special-purpose software components

The boundary between the standard and the dynamic AUTOSAR software is designed to pass through the SW-C level. In other words, AUTOSAR SW-Cs sandbox in the plug-ins, allowing them to interact with the rest of the system through standard SW-C ports, while the underlying concepts, such as the RTE, BSW and legacy ASW remain unchanged. Figure 1 gives an overview of how the plug-in concept is related to the built-in software. In the figure, dotted lines are used to show the plug-ins and their connections, whereas solid lines are used for the AUTOSAR SW-Cs and their links.

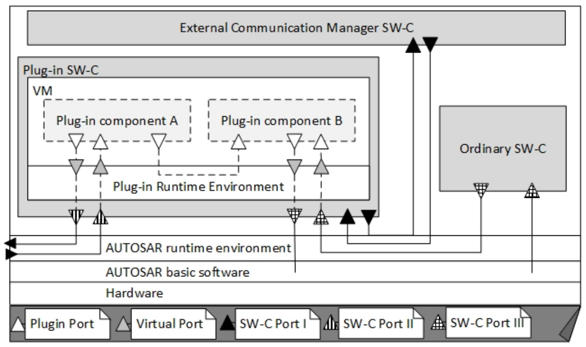

Figure 1: The dynamic component structure.

A special-purpose SW-C, called plug-in SW-C, refines ordinary AUTOSAR SW-Cs by embedding a virtual machine (VM), a dynamically evolving middleware, called Plug-in Runtime Environment (PIRTE), and a number of plug-in components. The VM is assigned its own memory, as well as computational and communication resources, which it in turn distributes to the plug-ins. This allows to execute the plug-ins under a best effort scheme, avoiding competition for resources with the built-in functionality.

Another type of dynamic SW-Cs, normally only present on one ECU, is the external communication manager (ECM) SW-C. It inherits from the plug-in SW-C and adds a communication module for interacting with the external world. It serves as a gateway for plug-in installation, allowing to download and distribute plug-in binaries to the different ECUs, as well as to transfer information to and from off-board services, e.g. for participating in FESs.

For the concept to work, OEMs must provide plug-in and ECM SW-Cs, which to start with only contain VMs and APIs in the form of provided and required SW-C ports, connected to the rest of the system through the RTE. The ECM SW-C should contain addressing information to a trusted server, containing plug-in databases. For safety reasons, the built-in software should monitor the exposed API and provide fault protection mechanisms for the critical signals.

3.1.2 Plug-in Runtime Environment

Similarly to the standard AUTOSAR concepts, plug-ins are accessed through provided and required plug-in ports, mediated by the PIRTE. Differently from the standard RTE, PIRTE contains both a static and a dynamic part. The static part consists of a mapping between the SW-C ports and the so-called virtual ports, which build up the actual static API available to the plug-ins. The dynamic part of the PIRTE is responsible for the installation and management of the plug-ins. Also, it allows dynamic configuration of the plug-in ports, connecting them either directly with other plug-in ports on the same SW-C, or with the virtual ports, for further transportation of the signals through RTE to the end recipient. This is done using context information, shipped with the binaries, and stored in the PIRTE.

A context typically consists of two parts, Port Initialization Context (PIC) and Port Linking Context (PLC). The PIC is used for communication between plug-ins on different SW-Cs and consists of data mapping between port names, defined by the plug-in developer, and SW-C unique port ids. The PLC describes the connections that should be established between the new plug-in ports and the virtual ports.

When a plug-in that is designed to communicate with the outside world is installed, an additional context message, the External Connection Context (ECC), is included into the installation package. An ECC is extracted by the ECM PIRTE and contains the location information of the external resource, e.g. its IP-address; the message id; and the internal routing information within the vehicle consisting of the id of the recipient ECU and the plug-in port.

3.1.3 Special-purpose ports

We distinguish between three types of SW-C ports, see Figure 1. While they look the same to the underlying RTE, they carry different types of data and are handled differently by the PIRTE. A pair of type I ports connect each plug-in SW-C with the ECM SW-C. Further, plug-in SW-Cs are connected to each other through pairs of type II SW-C ports, while type III ports are the typical AUTOSAR SW-C ports for the communication with the built-in software.

Internally, SW-C ports are made available to the plug-ins through a type-dependant mapping done in PIRTE to the so-called virtual ports. The mapping works also in the other direction, wrapping the plug-in data into correct format before outputting it to the SW-C ports.

Type I SW-C ports. There are several reasons for the plug-in SW-Cs to communicate with the external world, e.g. for plug-in installation, transfer of diagnostic messages, or for participation in a FES. This kind of communication is always handled by the ECM PIRTE, and relayed to the plug-in SW-Cs through type I SW-C ports.

During its initialization, the ECM PIRTE creates a socket client to set up a connection with a pre-defined trusted server. When new plug-in binaries arrive to the ECM from the server, they come together with a message type id (e.g. 0 for the installation package); the plug-in name; an id of the recipient plug-in SW-C; and a context describing the plug-in configuration for the particular vehicle. The ECM PIRTE extracts the id and uses it to write the binaries and the context on the appropriate type I SW-C port.

In the receiving SW-C, the plug-in PIRTE stores the installation package and processes the context. Once this process is finished successfully, PIRTE writes an acknowledgement message on its type I SW-C port, to be forwarded by the ECM to the trusted server.

Other kinds of external messages are relayed in the similar way. The only exception occurs when a message is destined to a plug-in in the ECM SW-C. In that case, the ECM PIRTE writes or reads directly to/from the plug-in port.

Type II SW-C ports. In the case of two plug-ins being located on the same SW-C, their ports are linked directly in PIRTE, according to the PLC information. In the other case, they are linked to the appropriate virtual ports, which map the sender port id to the id of the recipient plug-in port, again using the information extracted from the PLC. The recipient id is attached to the data before it is sent out on the type II SW-C port for further transportation through the RTE. On the receiving side, the procedure is reversed, the attached id is extracted, and the data is written to the plug-in port having the right id.

In such a way, any number of plug-in ports can communicate through one pair of static type II SW-C ports. Note that from the plug-in's perspective, its ports are normally mapped to other ports on the same SW-C, either plug-in or virtual, allowing the plug-in code to be unaffected by its allocation. Also, the plug-in and SW-C ports can have completely different formats, as long as the PIRTE is able to translate between these formats in its virtual ports.

Type III SW-C ports. Type III SW-C ports are used for communication with the built-in software, both BSW and ASW located in other non-plug-in SW-Cs. This is the simplest kind of ports from the PIRTE's perspective. Again, port connections are based on the PLC information, while virtual ports are used to transform between plug-in and AUTOSAR formats. However, no additional data is attached in such transformation.

3.2 Life Cycle Management of the plug-ins

For security reasons, all plug-in management is done through a pre-defined trusted server, which relieves the resource-constrained embedded system from much of the firewall issues. The address of the server is defined by the OEM and can only be altered by reprogramming the built-in software (the static part of the ECM PIRTE), which is a non-dynamic procedure with its own security mechanisms.

The server not only serves as a gateway for the plug-in binaries, but it is also responsible for verifying that new plug-ins are compatible with a particular vehicle configuration. Also, plug-ins may have dependencies on each other, which needs additional supervision. Thus, the trusted server acts as a central point of intelligence, performing compatibility checks and generating the different types of context that are

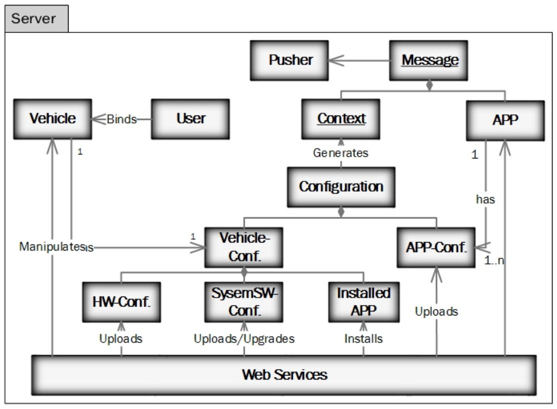

Figure 2: The structure of the trusted server

needed for successful plug-in configuration, see Section 3.1.

3.2.1 Structure of the trusted server

Figure 2 gives an overview of how the server is structured. On the top level from the users perspective, the server has a *User* and a *Vehicle* module, recording user profiles and associated vehicle details respectively. Each *Vehicle* has an associated configuration *Vehicle Conf*, which consists of hardware configuration in the *HW Conf* module, built-in software configuration in the *SystemSW Conf* module, and records of the installed plug-ins in the *InstalledAPP* module.

On the developer side, the *APP* module represents the application code stored on the server. Typically, an application consists of one or several plug-in binaries. Furthermore, each *APP* comes with one or several configurations (*SW conf*), which describe for various vehicle models how the plug-ins should be distributed in the vehicle and how the different plug-in ports should be connected. Combining the vehicle and plug-in configurations, the server creates a context for the specific combination of *APP* and *Vehicle*.

The trusted server provides two modules that communicate externally. The *Web Services* module presents an interface to the user of the server, whether it is the vehicle user, the OEM, or the plug-in developer. The *Pusher* module is used to interact with the vehicles through their ECM modules.

3.2.2 Plug-in operations

The server provides three kinds of operations through the *Web Services* module: user setup, upload of plug-ins and configurations, and plug-in (re)deployment. Except for typical settings, such as creating a user account, the user setup involves binding of a *Vehicle* module to a *User*, allowing the server to keep track of specific *Vehicle-User*-configurations.

Upload operations are typically done by OEMs or third party plug-in developers. OEMs should upload descriptions of the hardware resources that are available to plug-ins (*HW conf*), together with the exposed API, in terms of virtual ports in the available plug-in SW-Cs (*SystemSW conf*). Finally, the plug-in developers upload their binary code and descriptions of how to distribute their plug-ins on the available ECUs and how to connect the plug-in and virtual ports, based on the information in *HW conf* and *SystemSW conf*.

Plug-in installation is normally triggered by the user through a web portal. To begin with, the server checks whether the target vehicle meets the pre-requisites of the plug-in by comparing the vehicle configuration with the list of *SW conf* modules for the plug-in. If a match is found, i.e. there exists a description of how to distribute and connect the plug-ins in the vehicle, the turn comes to check the plug-in dependencies. In some cases, certain pre-requisite plug-ins must be installed in order for the new plug-ins to function. Conversely, the deployment operation can be hindered by an already installed plug-in being in conflict with the new plug-in functionality.

If the compatibility check fails, the server presents the reason for the failure to the user. If the check passes, the server creates a PIC context by assigning SW-C-scope unique ids to the plug-in ports, using the knowledge about the already installed plug-ins. Next, the port connection information, found in *SW conf*, is translated into a PLC context. Special care must be taken with the plug-in ports that will be connected to plug-ins located in other SW-Cs. In that case, the port ids of the recipient side must be included into the context that is communicated to the sending side SW-C. If any plug-in is designed to communicate externally, a package with ECC information is prepared for the ECM PIRTE.

Finally, the server extracts appropriate binaries from the *APP* database, combines them with the generated contexts into installation packages, adds the ids of the destination ECUs and the message types, and sends the packages to the target vehicle through the *Pusher* module. Afterwards, the server keeps track of the returning acknowledgement messages (acks) and records them into the *InstalledAPP* table.

When the uninstallation operation is invoked, the server starts by consulting its *InstalledAPP* table to check which plug-ins belong to this *APP* and whether there are some other installed plug-ins that are dependent on the plug-ins being uninstalled. If this is the case, the user is notified about the need to also uninstall the dependent plug-ins. Otherwise, uninstallation messages, containing the plug-in names and the ids of the ECUs on which they run, are sent to the vehicle and the *InstalledAPP* table is updated once successful uninstallation has been fully acknowledged.

Finally, the restore operation is used when an ECU hardware has been physically replaced, e.g. in a workshop. The server filters out previously installed plug-ins in the replaced ECU by querying the *InstalledAPP* module. Next, the usual installation steps are followed for each plug-in.

4. EXAMPLE APPLICATION

In parallel with the concept development, an open-source test platform was implemented to demonstrate the theoretical ideas. ARM-based Raspberry Pi (RPi) boards were used as representatives for the more powerful control units that are expected in the future. An open-source AUTOSAR implementation, ArcticCore [2], extended with a number of RPi-related drivers, see [17], served as the basic software. Each RPi unit was equipped with one plug-in SW-C running a Java virtual machine. Finally, the RPis were interconnected and one RPi was appointed as the ECM, responsible for the communication with the trusted server.

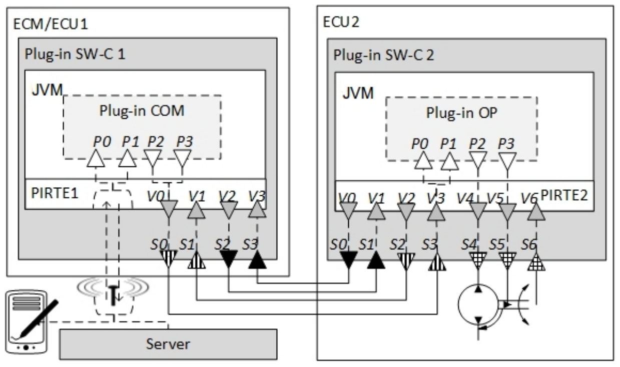

Figure 3: Overview of the example application.

In the following, an example application demonstrates the ideas presented in Section 3. The application enables a simple FES which connects a model car carrying two RPis to a smart phone and allows to remotely control the cars motion. It consists of two plug-ins, see Figure 3, the communicator (COM), located on the ECM RPi (ECU1) and listening to the signals from a smart phone, and the operator (OP), located on the other RPi (ECU2) and responsible for forwarding the actual control signals to the hardware.

The plug-in SW-Cs are connected in RTE by pairs of type I and type II ports, for exchanging external and plug-in messages respectively. SW-C2 provides an interface for the plug-ins to access basic software through three type III SW-C ports, translated into virtual ports *WheelsReq*, *SpeedReq*, and *SpeedProv* by PIRTE2 (V4-V6 in Figure 3). At start-up, PIRTE1 creates a connection with the trusted server.

Once the user triggers installation, the plug-ins are wrapped with necessary information, e.g. {0, 'COM', ECU1, com.pkg} and {0, 'OP', ECU2, op.pkg}, and sent to the ECM PIRTE. In its turn, it installs the COM-package, distributes the OP-package through its provided type I port, S2, and finally forwards the S3-port's ack message to the server.

The op.pkg contains the PIC, PLC and binaries for the OP plug-in. The PIC simply describes initialization of the plug-in ports with ids P0-P3, while PLC encodes connection configuration between the new plug-in ports and PIRTE's virtual ports. For example, assuming that P3 is a provided plug-in port that contains speed values, the *APP Conf* for this plug-in would indicate that it should be connected to the *SpeedReq* virtual port, which would be expressed by a {P3-V5}-post in the PLC. The whole PLC for OP of this example looks as follows: {P0-V3, P1-V3, P2-V4, P3-V5}. In the same way, the PLC of com.pkg defines the connections of the COM plug-in as {P0-, P1-, P2-V0.P0, P3-V0.P1}. Note the additional information in the last two PLC posts, notifying PIRTE1 that messages through these connections should be forwarded to ports P0 and P1 respectively on the recipient side, i.e. in the virtual port V3 of PIRTE2. The first two ports are not connected to any virtual port, meaning that PIRTE1 will communicate with them directly.

Since the COM plug-in is designed to communicate externally with a cell phone, its installation package also contains an ECC, in the following form: {{111.22.33.44:56789, ECU1, 'Wheels', P0}, {111.22.33.44:56789, ECU1, 'Speed', P1}}. The first post parts are used by PIRTE1 to set up a communication link to the cell phone, while the rest define the destination ports. When a messages arrive, depending on its id, 'Wheels' or 'Speed', it is written to P0 or P1.

The P0/P1 write operation triggers the COM software to perform its task. In this case, it simply formats and writes the data to its provided ports, P2 or P3, relaying the information to the associated virtual port, V0. Here, the destination plug-in port ids, P0 and P1 respectively, are appended and the contents are forwarded to S0. Next, RTE transfers the signals to S3 on SW-C2. These are mapped to V3, where the recipient ids are decoded and the signals are written on either P0 or P1. The OP code transforms the signals into appropriate calls to the basic software and finalizes the signal chain by writing to P2/P3, leading to a new signal in S4/S5 and a call to the underlying software. Note that there may exist unused virtual ports, such as V6 in SW-C2, which are set up by the OEM for the use of future plug-ins.

5. RELATED WORK

Several well-established component-based frameworks exist for embedded systems, such as Koala [13] and SaveCCM [8]. They are both similar to the AUTOSAR model in principle, and thus only provide static configuration of the system.

Other researchers have investigated the use of Java in AUTOSAR based systems, such as the KESO compiler [16]. However, this solution generates native code for each ECU, and is thus less suitable for dynamic reconfiguration.

Dynamic reconfiguration in automotive systems has been studied in the DySCAS project [1]. It defines a completely new architecture, with focus on mechanisms for self-reconfiguration. This however adds a lot of complexity to the architecture, which is avoided in our work with a more restricted and pragmatic approach based on plug-ins.

Outside the automotive domain, several component based systems with dynamic reconfiguration mechanisms have been reported. In [5], an approach is described that focuses on state-preserving updates of resource-constrained nodes. However, it is not based on a VM, but requires node-specific binaries to be generated. In [6], a dynamic adaptation strategy for resource constrained devices is presented, based on the notion of models@run.time. It assumes a mix of static and dynamic software components and allows to tune the dynamic ones to adapt the system behavior, e.g. memory usage. In difference, it does not offer a way to incorporate dynamic behaviour within a fully statical context, such as AUTOSAR.

In the avionics domain, the integrated modular avionics (IMA) architecture [15] strongly resembles the ideas of AUTOSAR. It aims at hardware/software decoupling, thus opening up for hardware optimization and a more flexible software development. However, the IMA systems will still be static between controlled upgrades, just as today's AUTOSAR systems are only updated (re-flashed) at service intervals.

A Java VM based approach is SEESCOA [14], which defines its own component model. It deals, among other things, with issues related to transferring the state of components from the old version to a new version during upgrades. This issue is dealt with more pragmatically in our approach, by mandating a plug-in to be stopped before being updated, and then restarted fresh.

The well-spread OSGi technology [11] defines a dynamic component architecture for Java-based applications, including life cycle management and security issues. Besides being restrictive in the choice of programming language, typical OSGi implementations tend to be too resource demanding for automotive applications. However, Concierge [12], an OSGi implementation tailored to embedded devices, could possibly be considered.

This work differs from all the above in that it provides dynamic installation of software components in an AUTOSAR based control system, executing the plug-ins in a Java VM. As described in [4], the success of FES is not only dependent on technology, but business relations are equally important, and key enablers are to have a solid architecture and good processes, methods, and tools.

6. CONCLUSIONS AND FUTURE WORK

This work pushes the limits of the classical AUTOSAR architecture, opening up for the installation of additional plug-in software in AUTOSAR based vehicles at runtime. In this paper, the main structural concepts that allow this shift to happen were outlined, roughly divided into vehicle internal and server side concepts.

The server structure defines the types of information that need to be provided by OEMs and plug-in developers to achieve a dynamic and modular plug-in management, adaptable to different vehicle platforms. It places high emphasis on the server as the main point of intelligence, somewhat relieving the vehicular system from the burdens of plug-in configuration and supervision.

On the vehicle side, the concepts of dynamic software components, plug-in runtime environment, and different kinds of software ports were proposed. These concepts allow to achieve a truly flexible and modular AUTOSAR plug-in environment, where plug-ins can be reallocated between the ECUs in a rather intuitive fashion, simply by reconfiguring their port connections. At the same time the legacy SW-C ports remain unchanged, allowing the underlying system to be static, while the plug-ins evolve dynamically.

This work opens up many directions for future research. Primarily, we intend to continue validating the results in the experimental platform, but also move it to an industrial setting and try it in a real vehicle. In addition, there is a need to investigate the characteristics of plug-in applications, by simply developing many more examples to discover what support is needed, both on the tool side and in the platform, to produce reliable quality plug-ins. In doing so, it will be important to find a safe way of handling dependencies and conflicts between different plug-ins. The possibility of using this technology for building FESs also contains many opportunities for researchers, to better understand how to actually build a well-functioning system-of-systems.

7. ACKNOWLEDGMENTS

This project is supported by Vinnova (grant no. 2012-02004), Volvo Cars, and the Volvo Group.

8. REFERENCES

[1] R. Anthony, et al. Towards a dynamically reconfigurable automotive control system architecture. In *Embedded System Design: Topics, Techniques and Trends.* Springer, 2007.

[2] ArcticCore product page. http://www.arccore.com. Accessed on 2013-11-12.

[3] J. Axelsson and A. Kobetski. On the conceptual design of a dynamic component model for reconfigurable autosar systems. In *5th Workshop on Adaptive and Reconfigurable Embedded Systems, Philadelphia*, 2013.

[4] J. Axelsson, E. Papatheocharous, and J. Andersson. Characteristics of software ecosystems for federated embedded systems: A case study. To appear in *Information and Software Technology Journal*, 2014.

[5] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In *Embedded System Design: Topics, Techniques and Trends.* Springer, 2007.

[6] F. Fouquet, et al. A dynamic component model for cyber physical systems. In *Proc. of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, 2012.

[7] S. Fürst, et al. Autosar–a worldwide standard is on the road. In *14th Intl VDI Congress Electronic Systems for Vehicles*, 2009.

[8] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Torngren. Saveccm-a component model for safety-critical real-time systems. In *Proc. of 30th IEEE Euromicro Conference*, 2004.

[9] A. Kobetski and J. Axelsson. Federated robust embedded systems: Concepts and challenges. Technical report, SICS Swedish ICT, 2012.

[10] The OSEK/VDX portal. http://portal.osek-vdx.org. Accessed on 2013-11-12.

[11] The OSGi alliance portal. http://www.osgi.org. Accessed on 2013-11-12.

[12] J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[13] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3), 2000.

[14] Y. Vandewoude and Y. Berbers. Run-time evolution for embedded component-oriented systems. In *Proc. of IEEE Intl Conf. on Software Maintenance*, 2002.

[15] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conf.*, 2007.

[16] C. Wawersich and I. T. M. Stilkerich. The use of java in the context of autosar 4.0. In *Embeded World*, 2011.

[17] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang. Porting an autosar-compliant operating system to a high performance embedded platform. In *3rd Embedded Operating Systems Workshop*, 2013.