# Architectural Concepts for Federated Embedded Systems

Jakob Axelsson
Swedish Institute of Computer Science (SICS)
SE-164 29 Kista
Sweden
+46 72 734 29 52
jakob.axelsson@sics.se

Avenir Kobetski
Swedish Institute of Computer Science (SICS)
SE-164 29 Kista
Sweden
+46 72 238 92 70
avenir.kobetski@sics.se

## ABSTRACT

Federated embedded systems (FES) is an approach for systems-of-systems engineering in the domain of cyber-physical systems. It is based on the idea to allow dynamic addition of plug-in software in the embedded system of a product, and through communication between the plug-ins in different products, it becomes possible to build services on the level of a federation of products. In this paper, architectural concerns for FES are elicited, and are used as rationale for a number of decisions in the architecture of products that are enabled for FES, as well as in the application architecture of a federation. A concrete implementation of a FES from the automotive domain is also described, as a validation of the architectural concepts presented.

## Categories and Subject Descriptors

D.2.11 [**Software Architecture**]: Domain-specific architectures.

## General Terms

Management, Documentation, Performance, Design, Reliability, Standardization, Verification.

## Keywords

Systems-of-systems; federated embedded systems; system architecture; cyber-physical systems.

## 1 INTRODUCTION

With the increasing availability of affordable communication services, the possibility to connect different systems to each other has grown in importance, and led to large interest from industry and academia in the challenges of creating s*ystems of systems* (SoS) [14]. Some characteristics of an SoS is that a number of independent systems are connected to create emergent functions and properties at the SoS level. Each constituent system, which will be referred to as a *product* in this paper, has a value on its own, even when used outside the SoS, and may be delivered and deployed independently by different manufacturers. The integration of the products into an SoS can use any kind of interface, including mechanical or electrical, but often the communication interfaces through wired or wireless connections are the most prominent ones.

Recently, SoS have also been given attention in the area of *cyber-physical systems* (CPS). Here, the traditional embedded systems (ES), where electronics and software of a product interact with the physical world through sensors and actuators, are extended with connectivity [5]. Due to their interaction with the physical world, CPS are often subject to other, and more stringent, requirements than other software-based systems, including dependability, timing requirements, product cost, and various life-cycle related qualities. For connected CPS, security also becomes an issue since the communication interfaces provide an entry point which could be subject to threat. The architecture of a typical CPS in a product is a complex system in itself, using a distributed architecture where a number of computer nodes are connected through internal communication networks.

In this paper, we will present findings related to a kind of SoS in the CPS area, which we call *federated embedded systems* (FES). In a FES, the creation of the SoS is based on connecting the ES in each product with each other, and also potentially with software running on servers outside the embedded systems. In this way, it becomes possible to create services on top of a combination of products. We call such an SoS a federation, since the constituent systems choose to participate voluntarily, for mutual benefit of the participants. The federation services are the intended emergent functions of the SoS. A single product may at any time participate in a number of federations, and the actual federations they form can vary dynamically over time, with participants coming and leaving based on their interest in the federation.

To provide a simple example of a FES from the transportation domain, consider a system that gives vehicles information about the status of the traffic lights ahead of them, allowing them to adapt their speed to pass without stopping. This has benefits both to each vehicle since it can lower fuel consumption, and to the road owners due to higher traffic throughput. The products involved would be the vehicles, and in addition, a connection to the traffic light controller is needed. Each vehicle could communicate its position, planned path, and speed, and receive information about the suggested speed to pass the next red light. This suggestion can either be displayed to the driver, or fed into the vehicle's cruise control, if it is lower than the set speed.

A key benefit of FES is that the adaptation of a particular product to a certain federation should be flexible and dynamic, allowing the addition of services that were not thought of at the time of designing the products, something that it is not possible with a pre-defined communication interface. In traditional ES, the example above would have required many years of standardization on the level of individual communication messages [19], and would only be implemented in new vehicles, whereas FES allows new services to be deployed also in already produced vehicles that just have the basic mechanism.

As described in [4], one of the key success factors in developing FES is the software architecture. In fact, there are two architectures that are relevant, namely the base *product* *architecture* whose ES will be enabled for participating in federations, and the *federation architecture*, that structures the services provided by that federation, which may need to include a large number of different products. The product architecture can be thought of as an infrastructure on which FES are built, whereas the federation architecture is the applications using the infrastructure.

Both these architectures have a number of challenges, some of them shared, and others individual. Therefore, the two research questions of this paper are as follows:

1. What are the important architectural characteristics needed to enable a product for FES?
2. What are the important architectural characteristics of a federation service to be built on products enabled for FES?

The main contribution of the paper is thus a description of important elements that could become a reference architecture for FES. It also includes an initial validation of these findings.

The research method used is based on design science [11] and constructive research [8], and it has been driven by interactions with practitioners in over 10 companies who in a sense represent stakeholders of different actors that are needed in an ecosystem around FES. The research has been highly iterative, but some of the main activities were:

1. *Applications and key concepts*: After formulating our initial ideas, a series of workshops were conducted with industry partners to brainstorm a large number of applications and also identify key concepts needed [13].
2. *Key stakeholders and concerns*: Based on the selected concept, a series of interviews were conducted with industry representatives, leading to the identification of a number of needs and concerns in the areas of business models, architecture, and process, methods and tools [4].
3. *Architecture design*: Given the requirements, the important parts in the architectures for FES were designed based on best practices for architecture descriptions, as documented in international standards [12].
4. *Implementation and validation*: To validate the architectures, a system based on it was developed. The application domain is automotive, and the implementation is based on the AUTOSAR industry standard [3].

This paper follows the structure of the research approach, with the first step covered in this introduction. In the next section, the key system concerns that drove the architecture are presented, followed in Section 3 by the architectural concepts. In Section 4, the implementation and validation is described, and in Section 5 some of the findings are discussed further. Finally, in Section 6, the conclusions are summarized together with some directions for further research.

## 2 ARCHITECTURAL CONCERNS

In this section, the main concerns of different stakeholders on the architecture are discussed. They will be presented as a set of qualities that are essential in FES. Some of the qualities are equally relevant for both the federation and product architectures, and others are primarily relevant for one of them (although there may be minor implications also for the other). The section also discusses some of the tradeoffs between the qualities. Since the concerns described here are valid for a general FES, a specific instance would normally have many other concerns that relate to its functions and its environment. Table 1 gives a summary of the concerns, and to which of the two architectures they primarily apply.

### 2.1 Dependability

The first concern is *dependability* [2], which applies to both architectures. For a federation service, the dependability is important since the services are what are offered to its users, and they will rely on the information provided by them.

For the products, they often have dependability requirements when used stand-alone, and the added mechanisms of plug-in software must not compromise their integrity, no matter what plug-ins are installed.

### 2.2 Security

*Security* is related to dependability [2]. It is essential that opening up the product architecture to federations does not lead to possibilities to inject malware, or access private information in the products. At the same time, the federation level functionality must also be protected against tampering and intrusion attempts.

### 2.3 Assurability

Closely related to dependability is *assurability*, i.e., to be able to efficiently and effectively deal with verification and validation. On the federation level, this means to be able to assure the qualities of the emergent services offered, which has to rely on information and processing from all the participants, and hence requires verification of parts of these.

For the products, it is mainly a question of verifying dependability concerns of the base product, given a set of plug-ins.

**Table 1. Summary of architectural concerns for FES.**

| Acronym | Concern | Product architecture | Federation architecture |
|---|---|---|---|
| D | Dependability | x | x |
| S | Security | x | x |
| A | Assurability | x | x |
| V | Variability | x | x |
| C | Composability | | x |
| P | Portability | x | x |
| O | Openness | x | |
| F | Flexibility | x | |
| R | Resource usage | x | x |
| M | Maintainability | | x |

### 2.4 Variability

For federation services, *variability* is given by the fact that many different product types could be potential members in the federation, and a there may be variations between them, so that different variants of the federation services are needed.

For the products, it is often the case that the same producer uses a product-line approach, having somewhat different architectures, to which the general mechanisms for participating in federations need to be adapted. In both cases, the architectures must support efficient ways of dealing with the variability.

## 2.5 Composability

For the federation services, a main concern is *composability*. A product should be allowed to participate in several federations simultaneously, and hence the services must be possible to develop independently and be used in parallel within the same ES. In cases of conflicts between services, these conflicts should be detectable before connecting to the services.

## 2.6 Portability

For both architectures, a key issue is *portability*. On the federation level, the services should be possible to execute on the hardware platforms in different products, to avoid the need of the federation developer to have development tools for many platforms.

In the product architecture, portability of the mechanisms that enable federations is essential, and the intrusion in the existing functionality of the control units should be minimized. Note that due to the distributed nature of many ES, it could sometimes be necessary to include federation mechanisms in several control units, to be able to access local data in them.

## 2.7 Openness

The product architecture should exhibit *openness* to federation services, allowing them to interact with the base application in a control unit and access data in it.

## 2.8 Flexibility

The product architecture should offer a high *flexibility* in what combinations of services can be installed, and in what interactions they may have with the base application. This is to not limit what federation services can be developed.

## 2.9 Resource usage

Most ES are resource constraint, since they are parts of products where the cost is an important business factor. Therefore, the *resource usage* of the federation mechanisms is important. Inevitably, a dynamic mechanism for handling various numbers of federations will require that additional processing power and memory is installed in the control units, but this should be minimized.

For the federation services, bandwidth for communication between different products and servers can be a limited resource.

## 2.10 Maintainability

When a federation is operating, plug-ins will be distributed over possibly a large number of products, and the *maintainability* of the federation must be assured. This includes handling updates of the software implementing federation services, managing situations where the product is repaired (e.g., when control unit hardware is replaced), and other life-cycle events.

## 2.11 Trade-offs

As usual, some of the architectural concerns are contradicting, and trade-offs are needed. In this case, one important trade-off is between *openness* on the one hand, and *dependability* and *security* on the other hand. The openness is providing value to system users in allowing the systems to participate in a wide range of federations, but at the same time it can lead to risks in disturbing the base product.

Another trade-off is between *flexibility* and *resource usage*. To allow maximum flexibility, generous processing resources should be provided in the control units, but that leads to higher product cost.

## 3 ARCHITECTURAL DESIGN

After having described the architectural concerns, this section will elaborate on key design decisions made in the product and federation architectures, with the rationale for those decisions expressed in relation to the concerns. In the next part of this section, some of the programming concepts are presented that were chosen to allow efficient development of FES. Then, in the remaining two subsections, the key concepts of the product architecture and federation architecture are described. The description is on the level of a reference architecture, since it deals with the concepts related to FES, whereas all concrete architecture instances would also include many other aspects specific to its functionality and domain. The main constructs in the architectures are shown in Figure 1, with indication of the rationale based on the concerns. In the figure, the shaded parts are belonging to the federation architecture and the rest belong to the product architecture. The letters in black circles indicate concerns, using the acronyms indicated in Table 1.

## 3.1 Programming concepts

To be able to implement federation services that integrate products and central servers, the functionality has to be expressed as a number of software modules that can be allocated to different parts in the federation. A key aspect of the architecture for FES is the idea of using plug-in software in the products, and this concept is described in the next subsection, followed by a presentation of the structuring mechanism for services.

### *3.1.1 Plug-in software*

In order to deal with the *flexibility* concerns, the ES in the products will be extended with a mechanism for plug-in software that can be added dynamically, in a way similar to how a smart phone can be extended with apps. Through the combination of connectivity and plug-in software, a large freedom is given to the service developers in where to allocate software, and potentially, much more data from the products can be accessed than would be possible if only having a fixed communication interface with no software in the ES. The *resource usage* of communication bandwidth would be reduced, since only those signals actually needed by a service are sent over the external interface. Also, it becomes possible to allocate functionality that closes local control loops directly in the product, without the need of going over slower external communication. To make a given product adapted for a certain federation, the correct plug-in modules have to be added, and it is possible to dynamically select which federations are relevant by adding or removing plug-ins during runtime.

A key benefit of FES is the flexible interfaces created through the plug-in mechanisms, which makes it possible to add services that were not thought of at the time of designing the products, Beyond the use in SoS, the plug-in mechanism also has other uses, for instance in shortening the deployment time of new features, and opening up the ES for open innovation by third party.

### *3.1.2 Component-based software*

The main idea about FES is to provide an efficient way of building SoS where the emergent functionality is described as federation services, and hence the programming concepts used to create those services are very important. Based on the *variability* and *composability* concerns, we advocate the use of a component-based approach [9] for structuring all parts of a federation service, including the plug-in software and software residing on servers. In such an approach, all functionality is encapsulated in components whose only relations to the outside are through their own ports.

To create a system, the components of different ports are connected. The data that is transferred over these ports will, in the context of FES, mainly be signals, which can be periodic or aperiodic, and due to the highly distributed nature of the system, the communication will be asynchronous.

In the description of a service, communication between components is ignorant of whether they are allocated as a plug-in to a product, or if they are on a server outside the ES, or a combination. Due to the general nature of these mechanisms, it is also possible for one federation to provide services through ports, to which another federation can connect and build new services on top of the ones provided by the first federation.

The components are all concurrent, and if they are plug-ins, they are started immediately when they are installed on the control unit, after which they stay alive until the unit is shut down. Internally, the components may implement various application-dependent states, but more generically, they will need to deal with the formation of federations, i.e., if the product is currently actively participating in a federation, or if it is only enabled for it through the installation of the relevant plug-ins.

## 3.2 Federation architecture components

Having defined the core FES concepts of plug-in software and connectivity, the federation level concepts will be studied next. The presentation is on the level of a number of functions that are needed for managing the operation of a federation and for managing its lifecycle, as illustrated in Figure 1.

### *3.2.1 Federation operation management functions*

In order to operate a federation, a number of functions are needed to coordinate the execution, connecting products, etc. These functions are to some extent application dependent, in that not all applications necessarily require all functions, or the full functionality, depending on the dynamic structure of the federation. Therefore, the presentation here is limited to listing a number of functions that may be needed.

The first set of functions relates to federation formation and dissolving. Once a federation has been created by setting up the necessary server-side software, and making plug-ins available to products, a given product has to *discover* the federation. This can mean different things, since a federation can only be relevant under certain conditions, such as a certain location. Next, the product needs to *join* the federation, to actually become an active member of it, after which the *operation* phase starts. During operation, the products need support for *addressing* specific other products in the federation to let them communicate. Finally, a product may choose to *leave* the federation, if it becomes irrelevant. For all these functions, application specific *rules* need to be described, and usually they will assign different *roles* to the members.

Another set of functions deals with overall management of the federation, including *supervision* of its operation, *fault handling*, *control*, *security mechanisms,* and *conflict handling* in case different participants try to take incompatible actions. These functions primarily address the concerns for *dependability* and *security*.

For all these functions, different possibilities exist when it comes

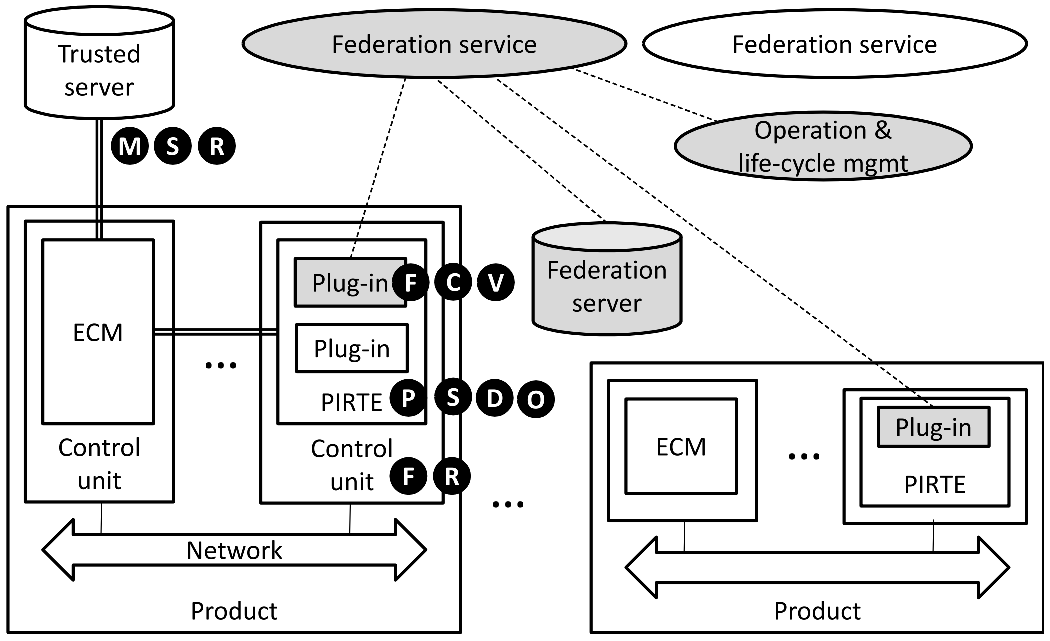

**Figure 1. Overview of FES architectures, with relations to concerns.**

to distribution of functionality to the members. One extreme case is complete centralization, where one server implements all the management functionality, and the other extreme is totally distributed, in which a federation consists only of products and no servers, and has the management functions included in plug-ins. Which solution is appropriate depends on the application characteristics of the federation services, and has to be defined in its specific architecture.

### 3.2.2 *Federation life-cycle management*

The main life-cycle phases of a federation are: development; configuration; operation; and maintenance. Most development aspects are covered in other parts of the paper, and the operation was covered in the previous subsection, so here the focus is on configuration and maintenance.

At the core of those phases is the handling of plug-ins in the product. In the FES architecture, this is centralized in a component called the *trusted server* [16], to efficiently deal with concerns for *maintainability* and *security,* but also to minimize *resource usage* in the ES. The trusted server is the only place from where a plug-in can be downloaded into a certain product, which means that many security mechanisms can be installed there instead of in the product. The trusted server stores all available plug-ins, and keeps track of which plug-ins are actually installed in a given product instance. It is thus responsible for maintenance activities, such as restoring plug-ins after a hardware repair of a control unit in the product, or upgrading to new versions.

## 3.3 Product architecture components

In this subsection, the elements that need to be included in the product architecture to deal with plug-in execution and communication are described, together with some tool support.

### 3.3.1 *External communication manager*

The communication with other systems is handled through the External Communication Manager (ECM) component. This includes both the external signaling of a plug-in which is part of a federation service, and the communication related to plug-in management.

There is only one instance of this component in each product, and it is typically located in the control unit that contains the wired or wireless communication interface. As shown in Figure 1, all communication related to plug-in management is between the ECM and the predefined trusted server, for reasons described above.

When a plug-in arrives in the ECM from the trusted server, the ECM will do three things: First it will examine what components are parts of the plug-in, and on which control units they will be distributed. Secondly, it will investigate the connections between those components, and ensure that appropriate addressing information is provided to implement the communication within the target architecture. Lastly, it will send each plug-in to its destination control unit, where it will start executing. Note that it is possible to analyze these steps offline on the trusted server, and thereby minimizing the intelligence needed in the ECM, which is often desirable to minimize *resource usage* in the ES. The control unit will acknowledge successful installation to the ECM, and inform the trusted server of the status for future maintenance actions.

In the case of a distributed product architecture, there are two ways in which the plug-in code can be stored. One option is to store it centrally in the control unit where the ECM is allocated. In that case, the plug-ins will be retransferred to the different control units on system startup, and then executed from RAM. If the control units have provision for local storage, e.g. in FLASH memory, another possibility is to store the code locally in each control unit, which would reduce start up time. However, this requires additional software mechanisms, such as a rudimentary file system, which is not usually available in ES control units.

### 3.3.2 *Plug-in runtime environment*

At the heart of a FES is the mechanism for plug-in execution in the product architecture. In our concept, this mechanism is encapsulated in a component called the Plug-In Runtime Environment (PIRTE), as shown in Figure 1. It consists of a virtual machine (VM) executing a machine independent representation of the plug-in software, thereby contributing to the *portability* of the plug-ins.

Typically, a producer would need to provision for plug-in execution in a number of control units based on different hardware. Therefore, the VM itself should be written in a high-level language to the greatest possible extend to increase its portability between control units, and should only rely on a bare minimum of hardware and operating system services.

PIRTE needs to have an interface to the underlying application software in the control unit, in order to access data and influence its behavior. To deal with *security* and *dependability* concerns, this interface is the only way in which a plug-in can access any parts of the control unit. To deal with *openness*, this interface is a trade-off point where product developers must consciously decide what input and output signals should be available to plug-ins. Possibly, different interface subsets could be offered, depending on the trust placed in the plug-ins.

The architecture does not make any assumptions about which control units of a distributed ES should be equipped with PIRTE, but it is possible to include as many PIRTE's as the designers find suitable. If several PIRTE's are included, there needs to be a logical communication channel between each of them, to allow communication between plug-ins that are part of the same federation service. Between each PIRTE component and the ECM component in the product, a logical communication link is also needed, to allow plug-in installation and management, and to allow plug-ins to communicate outside the product.

Also related to *dependability* is the need to isolate the VM from other application software in the control unit. PIRTE has to be allocated to its own operating system task, which should have lower priority than time critical application tasks, but should be assured a minimal share of the processing resources. Also, the VM will have its own memory area for stack and heap data. To make the product *flexible*, sufficient memory and processing resources need to be set aside for the PIRTE, while at the same time limiting the *resource usage* in order to keep cost down.

To be able to execute multiple plug-ins simultaneously, which is required for *composability*, the VM needs to be able to handle threads. In many ES, only statically defined tasks are possible, and the VM therefore has to provide its own threading concept.

### 3.3.3 *Configuration support*

As mentioned above, the product developers need to decide what signals should be available for the plug-ins on a certain control unit. If a component-based approach is used also for the control unit's built in functionality, which is sometimes the case [1][16], then it is possible to extract all available signals in the application.

Then, a simple, interactive approach can be used where the developer picks those signals that should be available to the plug-ins, and the generic PIRTE can be instantiated based on that, setting up all the required connections. This would address the *portability* and *variability* concerns for the product architecture. However, such a flexible approach may in practice lead to issues with unstable and volatile interfaces towards the plug-ins between product versions, which could negatively affect portability of plug-ins between product generations, and lead to increased variability on that side, if not used with care.

Even though complete tool support is possible for some parts of the interface, in particular in those cases where signals are only read by plug-ins, more elaborate efforts may be required for the parts of the interface that allow a plug-in to write signals to the product application. In particular, there is a risk that several independent plug-ins could try to write conflicting signals to the same interface, in which case an arbitration mechanism is needed to ensure *dependability*. At this point, we have not identified any general mechanisms to handle this, but it has to be dealt with based on the characteristics of the underlying application.

#### *3.3.4 Simulation*

To deal with *assurability* concerns, in a situation where the plug-ins are possibly developed by other organizations than the base products, it is necessary to provide tool support. If the actual product is not accessible to the plug-in developers, or if some testing cannot be performed efficiently, simulation support is needed, which captures the key characteristics of the product from the plug-in's point of view. This could be complemented with other kinds of static or formal verification.

## 4 INSTANTIATION AND VALIDATION

To validate the architectures, and provide a means to gather more empirical data on FES development and usage, a demonstrator has been created. It is called the Mobile Open Platform for Experimental Design (MOPED) [3], and consists of a model car in scale 1:10, which is equipped with a distributed computer system consisting of three control units based on Raspberry Pi hardware, and connected via Ethernet. Two of the control units execute software based on the AUTOSAR automotive software standard [1], and the third is based on Linux, which makes the configuration very representative of the software in a real vehicle. Each AUTOSAR node has various sensors and actuators, whereas the Linux node acts as a telematics unit responsible for external communication. It will now be described how the FES architectures have been instantiated in MOPED. The main focus will be on the product architecture, and some of the key design decisions made will be explained.

Since AUTOSAR uses a component-based approach, it was decided to base the component model for services on similar concepts, to maximize transparency between the built-in software and plug-ins. However, the base software is implemented in C, whereas it was decided to use Java for services, in order to make use of existing Java VMs inside the PIRTE. Therefore, a Java library with basic classes for ports and connectors was created and used for programming services.

The PIRTE is generated from configuration files that exist within the AUTOSAR framework. Those files, which are on XML format specified by the AUTOSAR standard and hence tool vendor independent, contain all the information about what ports exist on application software components in the control unit. The developer can thus easily select which ports to make visible for plug-in software in this PIRTE. In the model car, one PIRTE is created in each of the two AUTOSAR units, and the ECM is allocated to the Linux telematics unit, since that is where the external communication is placed. Since the Linux node has a file system, which is not the case for AUTOSAR nodes, the plug-ins are stored in the Linux node and transferred to the respective PIRTE on system start-up.

To complete the development environment, a simulator has also been constructed, that allows execution of plug-ins on a PC to test new functionality prior to deploying them in the ES.

For life-cycle management of the plug-ins, a trusted server has been implemented, that allows the users to select which plug-ins to install in the car. Developers can upload new plug-ins, handle variants for different platforms, and manage versions.

To create a federation service, such as the traffic light speed adaptation mentioned in the introduction, would require the implementation of different federation management functions on a server, that can inform plug-ins in each car about the next time the traffic light will switch. Joining and leaving this federation would be based on the location of the car, since information about the traffic light is really only relevant when the car is close to the light and travelling in that direction. The federation management functions such as supervision and fault-handling, are in this example fairly uncomplicated.

## 5 DISCUSSION

Through the work presented in this paper, a foundation has been laid for experimenting with SoS in the form of FES. We believe that we have captured many elements of a future reference architecture for such systems, but we also recognize that open issues exist, that can only be resolved through further investigations based on example applications. In this section, some of these issues will be discussed.

In the current work, initial attempts to build federation services have been described, including the management functions that are needed in federations. Most likely, more can be done in structuring these functions, e.g. in layers, and identifying recurring patterns and levels of functionality in order to provide blueprint solutions that can be used when creating a federation service. This can be complemented with other types of support, such as programming libraries with useful routines that are needed.

In the proposed FES architecture, a component-based approach is used, and we think there are strong arguments for this. In the case study, this was realized in Java, mainly due to the availability of VMs and the wide spread use of the language. At the moment, there is really no evidence that points at a need for a programming language of such complexity for this kind of applications. It would be relevant to look at what a minimalistic language would be, which could also lead to a minimal VM with even less requirements on resources for plug-in execution, thereby removing some of the barriers of adaptation in very small ES, such as sensor networks.

The architectures described in this paper are technical ones, but it is important to also recognize the relation to business architectures, as discussed in [18]. With the open principles of FES, it becomes relevant to study the software ecosystems that result around a product and around federation services. Some initial results around this have been identified [4], and it is clear that the success of the approach is closely related to the possibility of finding appropriate business models, leading to additional concerns that the architecture must support.

A question that is often raised is whether there is a need for standardization to make the FES approach work. Standardization would be beneficial, but is utopic since it requires a certain maturity of the application domain and usually takes years if not decades to accomplish. FES is targeted at domains that have not reached this stage, and where innovation is moving very fast. Therefore the focus has instead been on providing flexibility and variability in order to cater for new circumstances. The plug-in mechanism gives access to a potentially much broader interface to an ES than would a traditional signal based communication interface.

In the design of the FES architecture, steps have been taken to allow dependable systems. However, dependability can never be assured by this level of architecture alone, but always depends also on characteristics of the application. Further work is needed, again based on empirical evidence from concrete examples, to investigate if even better support for building dependable services, and for ensuring dependability in the individual products, can be provided in the architecture. This includes issues like resolving conflicts between plug-ins and services.

Related to dependability is also the possibility to build automatic control functionality in the services, such as the traffic light speed control sketched above. Due to the distributed, net centric nature of FES, parts of the functionality of a service will be central and parts in the plug-ins of different products. This means that some communication will inherently be subject to unpredictable delays, and also the execution of plug-ins will be time-variant, since it depends on what other plug-ins share the computational resources. These are all fundamental threats to building control functions, especially if there are dependability requirements, which is often the case. Most likely, support for dealing with this will be needed. One approach could be to add explicit time stamps on signals, in order to allow recipients to compensate for delays in communication and computation. Also, signals may come from sources whose trustworthiness is not known, and therefore, concepts for describing the quality of signals may be needed. This could be combined with supervision by the federation of the quality produced by different participants, in order to detect, and possibly isolate, products which are not behaving correctly. A strength of the FES architecture is that it gives the designer many options for allocating functionality between the products and central servers, and this can be used to deal with some of the timing issues, by allocating control loops close to the sensors and actuators inside the products instead of centrally, which would increase the use of communication with unpredictable latencies.

## 6 RELATED WORK

Although the engineering of SoS is a fairly new area, that still lacks systematic methods and proven solutions, some previous work exists with relation to the results presented in this paper. To start with, there are different definitions of types of SoS. In [14], three different types are included, namely “directed SoS” which are centrally managed with a common purpose; “collaborative SoS” which also have a common purpose but lacks central coordination; and “virtual SoS”, which lacks both these. In [10], the type “acknowledged SoS” is added, which is similar to directed ones, but still retain their own individual objectives as well and stress the cooperative nature of the constellation. The FES concept can in principle encompass all these four types, and it is a matter of how the services are defined. As [10] points out, acknowledged SoS are probably the most common variant, and this is likely to apply to FES as well.

The same reference also discusses principles for SoS, and among these are “using an architecture based on open systems and loose coupling”, which is exactly the principle followed for FES. Meilich [15] also discusses principles for SoS, with a focus on net centric environments and thus highly applicable for FES, concluding that flexibility, composability, and extensibility are important concerns. In his words, “capabilities that can be assembled or composed on-the-fly will be how effectiveness will be measured”, and this is taken care of by the plug-in mechanism in the FES product architecture. In [21], a number of research challenges for SoS architecture is listed, and several of them are factors that also went into the design of FES, such as resilience (which encompasses dependability), flexibility, agility, and modularity.

One of the underlying thoughts in the FES architecture was to come up with concepts that can scale and evolve without increasing the complexity of the framework itself, and this need was earlier observed also in [20]. They point out that standardized interfaces are beneficial for the evolvability of SoS, but at the same time recognize that this is not so realistic. In FES, the approach has been to device a stable mechanism for plug-in development, giving flexibility in the interfaces and providing a kind of interface layer, in the terminology of the reference. Other thoughts on the elements of SoS architecture are provided in [6], who proposes a network based solution with design-by-contract interfaces between the parts. FES uses a similar, but possibly more flexible, approach through component-based software. The authors also identify the need for control functions in the SoS, and present a case study from the defense domain.

In [6], the concrete architecture for a SoS satellite system is elaborated, but it appears to be primarily an example of a “directed SoS”, and does not give more general principles that could support an SoS reference architecture.

In comparison to the above examples, the FES approach presented in this paper is a general framework that can encompass many of the concepts previously presented. It also addresses many of the concerns identified by others through a light-weight and highly flexible approach with moderate assumptions about coordination and control of the SoS. The current work is fairly unique in that it aims at creating a reference architecture for a wide class of SoS. Many others are either speculating on SoS in general on a high level of abstraction, or working with a singular system example. We are systematically analyzing the properties of certain general mechanisms, and validating them through concrete implementation.

## 7 CONCLUSIONS

In this paper, an approached to systems of systems called federated embedded systems has been described. As described in the two research questions that has directed the work, the focus has been on the architecture of the products participating in the SoS, and on the architecture of the management functionality needed to operate a FES. The description of these architectures is on the level of a reference architecture, and to be able to validate them, an experimental platform from the automotive domain has been developed.

### 7.1 Future work

The development of the FES concept is still at an initial state. We believe that many of the important parts have been identified, but they need to be more detailed in order to reach the ultimate goal, which is to create and validate a stable reference architecture for FES. The sound approach to reaching this goal is to gain more experience from actually building FES, and use empirical evidence from this to extend the architecture descriptions. In particular, two kinds of cases will be used. First, more example applications will be implemented as federation services, and this will primarily be done using the experimental platform described above. From this, common patterns in federation architectures can be inferred, in particular for the federation management functions. Secondly, the product architecture will be validated through case studies at partner companies, to check that the architectural solutions presented here can match a wide variety of product architectures.

An especially important area of future work is on assurances, and an overarching concern for all SoS is how to build reliable systems out of unreliable components. Better methods are needed to discover and handle conflicts between plug-ins, and to assess that timing behavior meets the requirements. In particular, it would be valuable to find ways of analyzing these effects on the trusted server, prior to installation of plug-ins. Since the trusted server keeps a record of what plug-ins are installed on a particular product, it has the needed information to analyze the consequences of adding one more plug-in. For instance, it could assess if there is sufficient memory and processing power available, but also check for conflicts with the other plug-ins.

## 8 ACKNOWLEDGMENTS

This research was funded by VINNOVA, the Swedish Agency for Innovation Systems (Grants No. 2012-02004 and 2012-03782), and was also supported in different ways by a large number of Swedish companies.

## 9 REFERENCES

[1] AUTOSAR. www.autosar.org.

[2] Avizienis, A. et al. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*. 1, 1 (Jan. 2004), 11–33.

[3] Axelsson, J. et al. 2014. MOPED : A Mobile Open Platform for Experimental Design of Cyber-Physical Systems. *Euromicro SEAA* (2014).

[4] Axelsson, J. et al. 2014. Characteristics of software ecosystems for Federated Embedded Systems: A case study. *Information and Software Technology*. (Apr. 2014).

[5] Broy, M. and Schmidt, A. 2014. Challenges in Engineering Cyber-Physical Systems. *Computer*. 47, 2 (2014), 70–72.

[6] Butterfield, M.L. et al. 2008. A system-of-systems engineering GEOSS: Architectural approach. *IEEE Systems Journal*. 2, 3 (2008), 321–332.

[7] Caffall, D.S. and Michael, J.B. 2005. Architectural framework for a system-of-systems. *2005 IEEE International Conference on Systems, Man and Cybernetics*. 2, (2005).

[8] Crnkovic, G.D. 2010. Constructive Research and Info-computational Knowledge Generation. *Model-Based Reasoning in Science and Technology*. L. Magnani et al., eds. 359–380.

[9] Crnkovic, I. and Larsson, M. 2002. *Building Reliable Component-Based Software Systems*. Artech House, Inc.

[10] Department of Defense. Systems Engineering Guide for Systems of Systems, v. 1.0, 2008.

[11] Hevner, A.R. et al. 2004. Design Science in Information Systems Research. *MIS Quarterly*. 28, 1 (2004), 75–105.

[12] ISO/IEC/IEEE Std. 42010. Systems and software engineering — Architecture description, 2011.

[13] Kobetski, A. and Axelsson, J. 2012. Federated robust embedded systems: Concepts and challenges. SICS Technical Report T2012:05.

[14] Maier, M.W. 1998. Architecting principles for systems-of-systems. *Systems Engineering*. 1, 4 (1998), 267–284.

[15] Meilich, A. 2006. System of systems (SoS) engineering architecture challenges in a net centric environment. *2006 IEEE/SMC International Conference on System of Systems Engineering*. (2006).

[16] Ni, Z. et al. 2014. Design and Implementation of a Dynamic Component Model for Federated AUTOSAR Systems. *Design Automation Conference* (2014).

[17] Ommering, R. van et al. 2000. The Koala Component Model for Consumer Electronics Software. *Computer*. 33, 3 (2000), 78–85.

[18] Papatheocharous, E. et al. 2013. Issues and challenges in ecosystems for federated embedded systems. *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems - SESoS '13* (New York, New York, USA, Jul. 2013), 21–24.

[19] SAE Std. J2735. Dedicated Short Range Communications (DSRC) Message Set Dictionary, 2009.

[20] Selberg, S.A. and Austin, M.A. 2008. Toward an evolutionary system of systems architecture. *18th Annual International Symposium of the International Council on Systems Engineering, INCOSE 2008* (2008), 2394–2407.

[21] Valerdi, R. et al. 2008. A research agenda for systems of systems architecting. *International Journal of System of Systems Engineering*.