Document Title	Explanation of Adaptive Platform Design
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	706

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-10

Document Change History

Date	Release	Changed by	Description
2018-10-31	18-10	AUTOSAR Release Management	Changes to reflect the latest SWS contents.
2018-03-29	18-03	AUTOSAR Release Management	Update of a logical view of AP architecture. Addition of Update and Configuration Management, State Management, Time Synchronization, Adaptive Network Management, Identity Access Management, Cryptography, and Core types.
2017-10-27	17-10	AUTOSAR Release Management	Added RESTful Communication
2017-03-31	17-03	AUTOSAR Release Management	Initial release

Disclaimer

Table of Contents

1 Introduction to this document

1.1 Contents

This specification describes the AUTOSAR Adaptive Platform (AP) design. The purpose of this document is to provide an overview of AP but is not to detail all the elements of AP design. It is to provide the overall design of the AP and key concepts for both AP users and AP implementers.

The document is organized as follows. It starts with Technical Scope and Approach to provide some background of AP, followed by Architecture describing both logical and physical views of AP. Independent chapters of Methodology and Manifest and all Functional Clusters follow, which are the units of functionalities of AP, each containing its overview and introductions to their key concepts.

The detailed specification and discussions on the explained concepts are defined in the relevant RS, SWS, TR and EXP documents.

1.2 Prereads

This document is one of the high-level conceptual documents of AUTOSAR. Useful pre-reads are [1] [2] [3].

1.3 Relationship to other AUTOSAR specifications

Refer to Contents and Prereads.

2 Technical Scope and Approach

2.1 Overview – a landscape of intelligent ECUs

Traditionally ECUs mainly implement functionality that replaces or augments electro-mechanical systems. Software in those deeply-embedded ECUs controls electrical output signals based on input signals and information from other ECUs connected to vehicle network. Much of the control software is designed and implemented for the target vehicle and does not change significantly during vehicle lifetime.

New vehicle functions, such as highly automated driving, will introduce highly complex and computing resource demanding software into the vehicles and must fulfill strict integrity and security requirements. Such software realizes functions, such as environment perception and behavior planning, and integrates the vehicle into external backend and infrastructure systems. The software in the vehicle needs to be updated during the lifecycle of the vehicle, due to evolving external systems or improved functionality.

The **AUTOSAR Classic Platform (CP)** standard addresses the needs of deeply-embedded ECUs, while the needs of ECUs described above cannot be fulfilled. Therefore, AUTOSAR specifies a second software platform, the **AUTOSAR Adaptive Platform (AP)**. AP provides mainly high-performance computing and communication mechanisms and offers flexible software configuration, e.g. to support software update over-the-air. Features specifically defined for the CP, such as access to electrical signals and automotive specific bus systems, can be integrated into the AP but is not in the focus of standardization.

2.2 Technology Drivers

There are two major groups of technology drivers behind. One is Ethernet, and the other is processors.

The ever-increasing bandwidth requirement of the on-vehicle network has led to the introduction of Ethernet, that offers higher bandwidth and with switched networks, enabling more efficient transfer of long messages, point-to-point communications, among others, compared to the legacy in-vehicle communication technologies such as CAN. The CP, although it supports Ethernet, is primarily designed for the legacy communication technologies, and it has been optimized for such, and it is difficult to fully utilize and benefit from the capability of Ethernet-based communications.

Similarly, performance requirements for processors have grown tremendously in recent years as vehicles are becoming even more intelligent. Multicore processors are already in use with CP, but the needs for the processing power calls for more than multicore. Manycore processors with tens to hundreds of cores, GPGPU (General Purpose use of GPU), FPGA, and dedicated accelerators are emerging, as

these offer orders of magnitudes higher performance than the conventional MCUs. The increasing number of cores overwhelms the design of CP, which was originally designed for a single core MCU, though it can support multicore. Also, as the computing power swells, the power efficiency is already becoming an issue even in data centers, and it is in fact much more significant for these intelligent ECUs. From semiconductor and processor technologies point of view, constrained by Pollack's Rule, it is physically not possible to increase the processor frequency endlessly and the only way to scale the performance is to employ multiple (and many) cores and execute in parallel. Also, it is known that the best performance-per-watt is achieved by a mix of different computing resources like manycore, co-processors, GPU, FPGA, and accelerators. This is called heterogeneous computing – which is now being exploited in HPC (High-Performance Computing) - certainly overwhelms the scope of CP by far.

It is also worthwhile to mention that there is a combined effect of both processors and faster communications. As more processing elements are being combined in a single chip like manycore processors, the communication between these processing elements is becoming orders of magnitude faster and efficient than legacy inter-ECU communications. This has been made possible by the new type of processor inter-connect technologies such as Network-on-Chip (NoC). Such combined effects of more processing power and faster communication within a chip also prompts the need for a new platform that can scale over ever-increasing system requirements.

2.3 Adaptive Platform – Characteristics

The characteristic of AP is shaped by the Overview – landscape of intelligent ECUs and Technology Drivers. The landscape inevitably demands significantly more computing power, and the technologies trend provides a baseline of fulfilling such needs. However, the HPC in the space of safety-related domain while power and cost efficiencies also matter, is by itself imposes various new technical challenges.

To tackle them, AP employs various proven technologies traditionally not fully exploited by ECUs, while allowing maximum freedom in the AP implementation to leverage the innovative technologies.

C++

From top-down, the applications can be programmed in C++. It is now the language of choice for the development of new algorithms and application software in performance critical complex applications in the software industry and in academics. This should bring faster adaptations of novel algorithms and improve application development productivity if properly employed.

SOA

To support the complex applications, while allowing maximum flexibility and scalability in processing distribution and compute resource allocations, AP follows service-oriented-architecture (SOA). The SOA is based on the concept that a system consists of a set of services, in which one may use another in turn, and applications

that use one or more of the services depending on its needs. Often SOA exhibits system-of-system characteristics, which AP also has. A service, for instance, may reside on a local ECU that an application also runs, or it can be on a remote ECU, which is also running another instance of AP. The application code is the same in both cases – the communication infrastructure will take care of the difference providing transparent communication. Another way to look at this architecture is that of distributed computing, communicating over some form of message passing. At large, all these represent the same concept. This message passing, communication-based architecture can also benefit from the rise of fast and high-bandwidth communication such as Ethernet.

Parallel processing

The distributed computing is inherently parallel. The SOA, as different applications use a different set of services, shares this characteristic. The advancement or manycore processors and heterogeneous computing that offer parallel processing capability offer technological opportunities to harness the computing power to match the inherent parallelism. Thus, the AP possesses the architectural capability to scale its functionality and performance as the manycore-heterogeneous computing technologies advance. Indeed, the hardware and platform interface specification are only parts of the equation, and advancements in OS/hypervisor technologies and development tools such as automatic parallelization tools are also critical, which are to be fulfilled by AP provider and the industry/academic eco-system. The AP aims to accommodate such technologies as well.

Leveraging existing standard

There is no point in re-inventing the wheels, especially when it comes to specifications, not implementations. As with already described in C++, AP takes the strategy of reusing and adapting the existing open standards, to facilitate the faster development of the AP itself and benefiting from the eco-systems of existing standards. It is, therefore, a critical focus in developing the AP specification not to casually introduce a new replacement functionality that an existing standard already offers. For instance, this means no new interfaces are casually introduced just because an existing standard provides the functionality required but the interface is superficially not easy to understand.

Safety and security

The systems that AP targets often require some level of safety and security, possibly at its highest level. The introduction of new concepts and technologies should not undermine such requirements although it is not trivial to achieve. To cope with the challenge, AP combines architectural, functional, and procedural approaches. The architecture is based on distributed computing based on SOA, which inherently makes each component more independent and free of unintended interferences, dedicated functionalities to assist achieving safety and security, and guidelines such as C++ coding guideline, which facilitates the safe and secure usage of complex language like C++, for example.

Planned dynamics

The AP supports the incremental deployment of applications, where resources and communications are managed dynamically to reduce the effort for software development and integration, enabling short iteration cycles. Incremental deployment also supports explorative software development phases.

For product delivery, AP allows the system integrator to carefully limit dynamic behavior to reduce the risk of unwanted or adverse effects allowing safety qualification. Dynamic behavior of an application will be limited by constraints stated in the Execution manifest. The interplay of the manifests of several applications may cause that already at design time. Nevertheless, at execution time dynamic allocation of resources and communication paths are only possible in defined ways, within configured ranges, for example.

Implementations of an AP may further remove dynamic capabilities from the software configuration for production use. Examples of planned dynamics might be:

- Pre-determination of service discovery process
- Restriction of dynamic memory allocation to the startup phase only
- Fair scheduling policy in addition to priority-based scheduling
- Fixed allocation of processes to CPU cores
- Access to pre-existing files in the file-system only
- Constraints for AP API usage by Applications
- Execution of authenticated code only

Agile

Although not directly reflected in the platform functionalities, the AP aims to be adaptive to different product development processes, especially agile based processes. For agile based development, it is critical that the underlying architecture of the system is incrementally scalable, with the possibility of updating the system after its deployment. The architecture of AP should allow this. As the proof of concept, the AP specification itself and the demonstrator, the demonstrative implementation of AP, are both developed with Scrum.

2.4 Integration of Classic, Adaptive and Non-AUTOSAR ECUs

As described in previous sections, AP will not replace CP or Non-AUTOSAR platforms in IVI/COTS. Rather, it will interact with these platforms and external backend systems such as road-side infrastructures, to form an integrated system (Figure 2-1 Exemplary deployment of different platforms, and Figure 2-2 Exemplary interactions of AP and CP). As an example, CP already incorporates SOME/IP, which is also supported by AP, among other protocols.

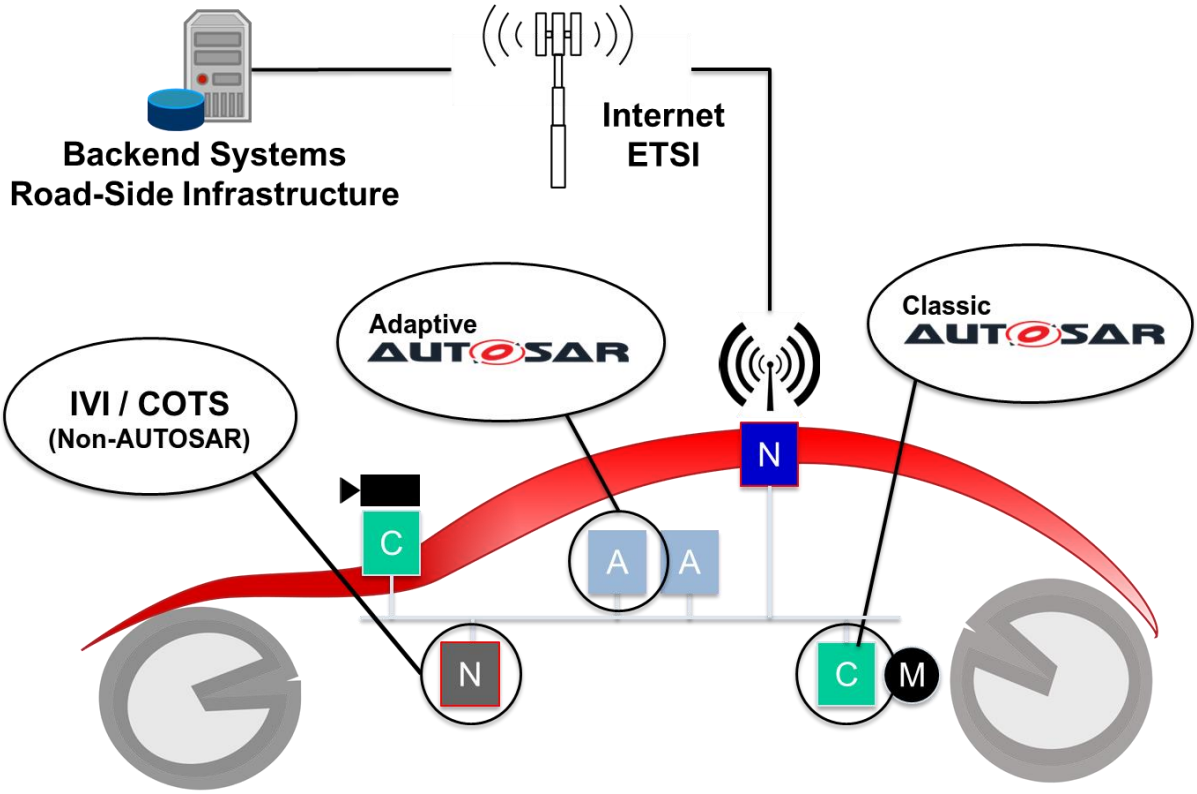

Figure 2-1 Exemplary deployment of different platforms

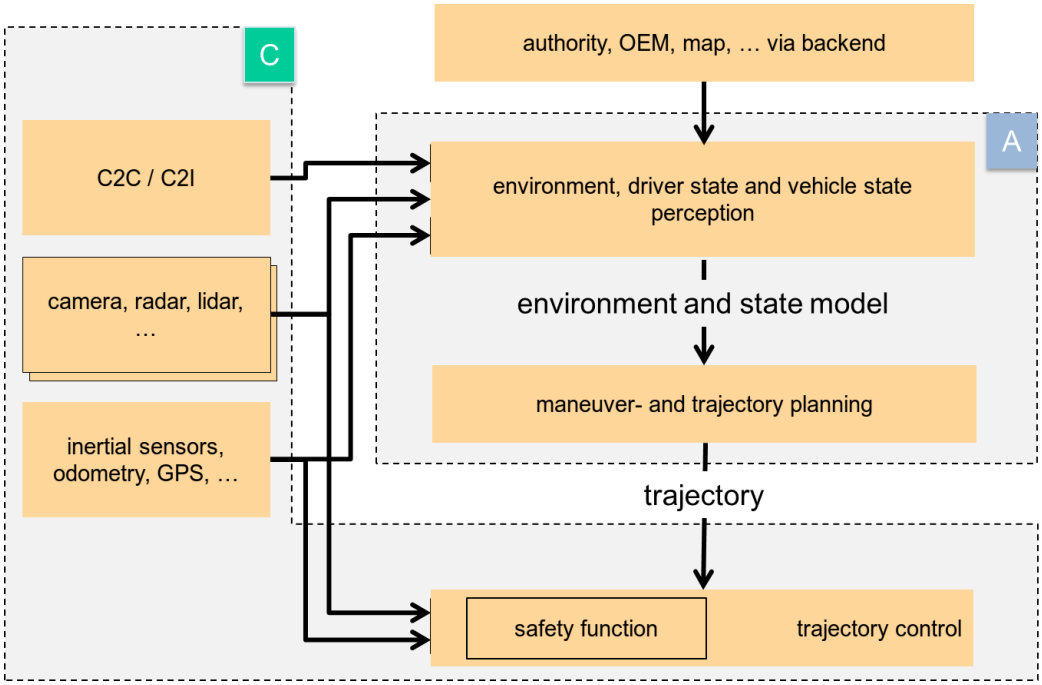

Figure 2-2 Exemplary interactions of AP and CP

2.5 Scope of specification

AP defines the runtime system architecture, what constitutes a platform, and what functionalities and interfaces it provides. It also defines machine-readable models that are used in the development of such a system. The specification should provide necessary information on developing a system using the platform, and what needs to be met to implement the platform itself.

3 Architecture

3.1 Logical view

ARA

Figure 3-1 AP architecture logical view shows the architecture of AP. The **Adaptive Applications (AA)** run on top of **ARA, AUTOSAR Runtime for Adaptive applications**. ARA consists of application interfaces provided by **Functional Clusters**, which belong to either **Adaptive Platform Foundation** or **Adaptive Platform Services**. Adaptive Platform Foundation provides fundamental functionalities of AP, and **Adaptive Platform Services** provide platform standard services of AP. Any AA can also provide Services to other AA, illustrated as **Non-platform service** in the figure.

The interface of Functional Clusters, either they are those of Adaptive Platform Foundation or Adaptive Platform Services, are indifferent from AA point of view – they just provide specified C++ interface or any other language bindings AP may support in future. There are indeed differences under the hood. Also, note that underneath the ARA interface, including the libraries of ARA invoked in the AA contexts, may use other interfaces than ARA to implement the specification of AP and it is up to the design of AP implementation.

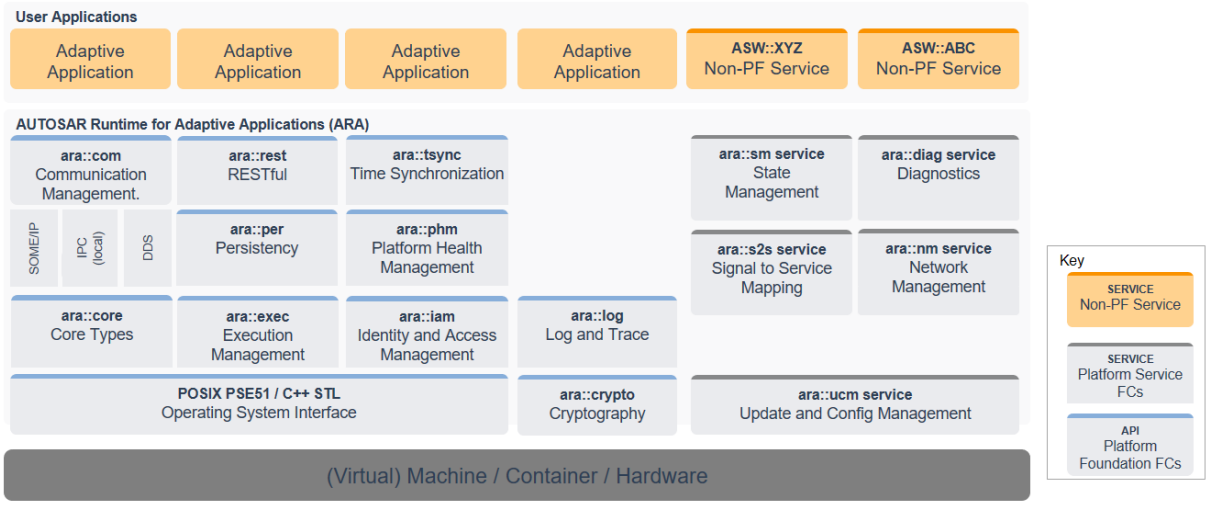

Figure 3-1 AP architecture logical view

Be aware that Figure 3-1 AP architecture logical view contains Functional Clusters that are not part of the current release of AP, to provide a better idea of overall structure. Further new Functional Clusters not shown here may well be added future releases of AP.

Language binding, C++ Standard Library, and POSIX API

The language binding of these API is based on C++, and the C++ Standard library is also available as part of ARA. Regarding the OS API, only PSE51 interface, a single-

process profile of POSIX standard is available as part of ARA. The PSE51 has been selected to offer portability for existing POSIX applications and to achieve freedom of interference among applications.

Note that the C++ Standard Library contains many interfaces based on POSIX, including multi-threading APIs. It is recommended not to mix the C++ Standard library threading interface with the native PSE51 threading interface to avoid complications. Unfortunately, the C++ Standard Library does not cover all the PSE51 functionalities, such as setting a thread scheduling policy. In such cases, the combined use of both interfaces may be necessary.

Application launch and shutdown

Lifecycles of applications are managed by Execution Management (EM). Loading/launching of an application is managed by using the functionalities of EM, and it needs appropriate configuration at system integration time or at runtime to launch an application. In fact, all the Functional Clusters are applications from EM point of view, and they are also launched in the same manner, except for EM itself. Figure 3-2 Applications illustrates different types of applications within and on AP.

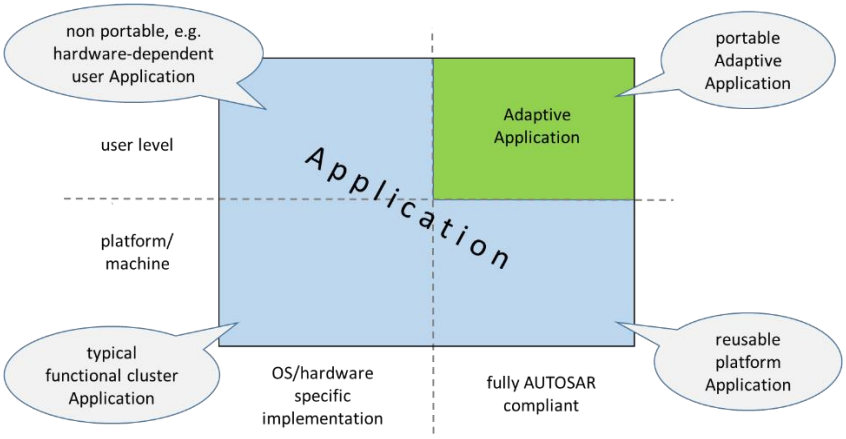

Figure 3-2 Applications

Note that decisions on which and when the application starts or terminates are not made by EM. Another FC, called State Management (SM), is the controller, commanding EM based on the design of a system, arbitrating different states thus controlling the overall system behavior. The SM also interact with other FCs to coordinate the overall machine behavior.

Application interactions

Regarding interaction between AAs, PSE51 do not include IPC (Inter-Process-Communication), so there is no direct interface to interact between AAs. The Communication Management (CM) is the only explicit interface. CM also provides Service Oriented Communication for both intra-machine and inter-machine, which are transparent to applications. CM handles routing of Service requests/replies regardless of the topological deployment of Service and client applications. Note that other ARA interfaces may internally trigger interactions between AAs, however, this

is not an explicit communication interface but just a byproduct of functionalities provided by the respective ARA interfaces.

Non-standard interfaces

AA and Functional Clusters may use any non-standard interfaces, provided that they do not conflict with the standard AP functionalities and also that they conform to the safety/security requirements of the project. Unless they are pure application local runtime libraries, a care should be taken to keep such use minimal, as this will impact the software portability onto other AP implementations.

3.2 Physical view

The physical architecture[1] of AP is discussed here. Note that most of the contents in this section are for illustration purpose only, and do not constitute the formal requirement specification of AP, as the internals of AP is implementation-defined. Any formal requirement on the AP implementation is explicitly stated.

OS, processes, and threads

The AP Operating System is **required** to provide multi-process POSIX OS capability. Each AA is implemented as an independent process, with its own logical memory space and namespace. Note that a single AA may contain multiple processes, and this may be deployed onto a single AP instance or distributed over multiple AP instances. From the module organization point of view, each process is instantiated by OS from an executable. Multiple processes may be instantiated from a single executable. Also, AA may constitute multiple executables.

Functional Clusters are also typically implemented as processes. A Functional Cluster may also be implemented with a single process or multiple (sub) processes. The Adaptive Platform Services and the non-platform Services are also implemented as processes.

All these processes can be a single-threaded process or a multi-threaded process. However, the OS API they can use differs depending on which logical layer the processes belong to. If they are AAs running on top of ARA, then they should only use PSE51. If a process is one of the Functional Clusters, it is free to use any OS interface available.

In summary, from the OS point of view, the AP and AA forms just a set of processes, each containing one or multiple threads – there is no difference among these processes, though it is up to the implementation of AP to offer any sort of partitioning. These processes do interact with each other through IPC or any other OS functionalities available. Note that AA processes, may not use IPC directly and can only communicate via ARA.

[1] The ‘physical architecture’ here means mainly the Process View, Physical View, and some Development View as described in [6].

Library-based or Service based Functional Cluster implementation

As in Figure 3-1 AP architecture logical view, a Functional Cluster can be an Adaptive Platform Foundation module or an Adaptive Platform Service. As described previously, these are generally both processes. For them to interact with AAs, which are also processes, they need to use IPC. There are two alternative designs to achieve this. One is “Library-based” design, in which the interface library, provided by the Functional Cluster and linked to AA, calls IPC directly. The other is “Service-based” design, where the process uses Communication Management functionality and has a Server proxy library linked to the AA. The proxy library calls Communication Management interface, which coordinates IPC between the AA process and Server process. Note it is implementation-defined whether AA only directly performs IPC with Communication Management or mix with direct IPC with the Server through the proxy library.

A general guideline to select a design for Functional Cluster is that if it is only used locally in an AP instance, the Library-based design is more appropriate, as it is simpler and can be more efficient. If it is used from other AP instance in a distributed fashion, it is advised to employ the Service-based design, as the Communication Management provides transparent communication regardless of the locations of the client AA and Service. Functional Clusters belonging to Adaptive Platform Foundation are “Library-based” and Adaptive Platform Services are “Service-based” as the name rightly indicate.

Finally, note that it is allowed for an implementation of an FC to not to have a process but realize in the form of a library, running in the context of AA process, as long as it fulfills the defined RS and SWS of the FC. In this case, the interaction between an AA and the FC will be regular procedure call instead of IPC-based as described previously.

The interaction between Functional Clusters

In general, the Functional Clusters may interact with each other in the AP implementation-specific ways, as they are not bound to ARA interfaces, like for example PSE51, that restricts the use of IPC. It may indeed use ARA interfaces of other Functional Clusters, which are `public` interfaces. One typical interaction model between Functional Clusters is to use `protected` interfaces of Functional Clusters to provide privileged access required to achieve the special functionalities of Functional Clusters.

Also, from AP18-03, a new concept of Inter-Functional-Cluster (IFC) interface has been introduced. It describes the interface an FC provides to other FCs. Note that it is not part of ARA, nor does it constitute formal specification requirements to AP implementations. These are provided to facilitate the development of the AP specification by clarifying the interaction between FCs, and they may also provide better architectural views of AP for the users of AP specification. The interfaces are described in the Annex of respective FC SWS.

Machine/hardware

The AP regards a hardware it runs on as a **Machine**. The rationale behind that is to achieve a consistent platform view regardless of any virtualization technology which might be used. The Machine might be a real physical machine, a fully-virtualized machine, a para-virtualized OS, an OS-level-virtualized container or any other virtualized environment.

On a hardware, there can be one or more Machines, and only a single instance of AP runs on a machine. It is generally assumed that this 'hardware' includes a single chip, hosting a single or multiple Machines. However, it is also possible that multiple chips form a single Machine if the AP implementation allows it.

3.3 Methodology and Manifest

The support for distributed, independent, and agile development of functional applications requires a standardized approach to the development methodology. AUTOSAR adaptive methodology involves the standardization of **work products** for the description of artifacts like services, applications, machines, and their configuration; and the respective **tasks** to define how these work products shall interact to achieve the exchange of design information for the various activities required for the development of products for the adaptive platform.
Figure 3-3 illustrates a draft overview of how adaptive methodology might be implemented. For the details of these steps see [3].

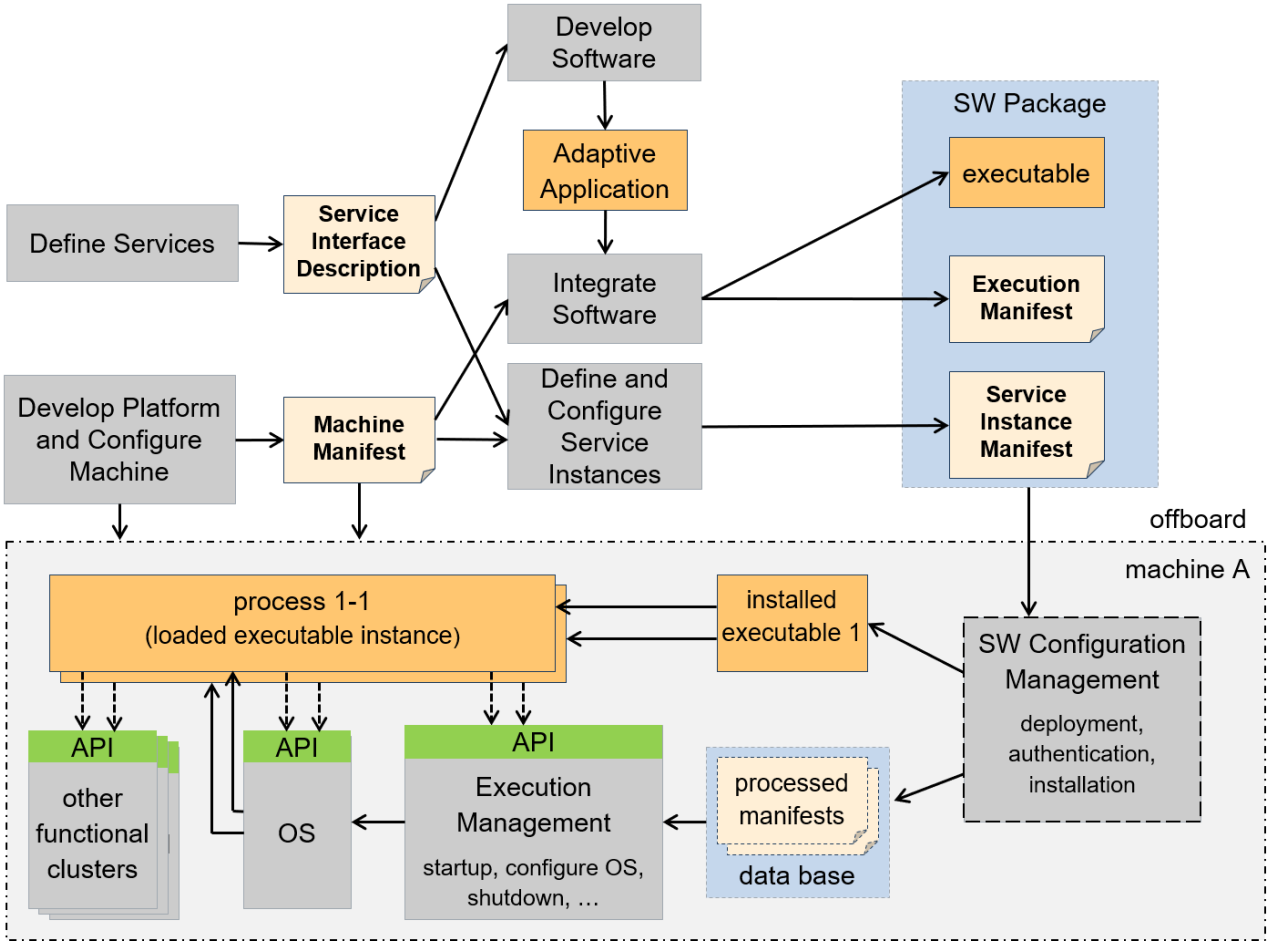

Figure 3-3 AP development workflow

3.4 Manifest

A Manifest represents a piece of AUTOSAR model description that is created to support the configuration of an AUTOSAR AP product and which is uploaded to the AUTOSAR AP product, potentially in combination with other artifacts (like binary files) that contain executable code to which the Manifest applies.

The usage of a Manifest is limited to the AUTOSAR AP. This does not mean, however, that all ARXML produced in a development project that targets the AUTOSAR AP is automatically considered a Manifest.
In fact, the AUTOSAR AP is usually not exclusively used in a vehicle project.

A typical vehicle will most likely be also equipped with a number of ECUs developed on the AUTOSAR CP and the system design for the entire vehicle will, therefore, have to cover both – ECUs built on top of the AUTOSAR CP and ECUs created on top of the AUTOSAR AP.

In principle, the term Manifest could be defined such that there is conceptually just one "Manifest" and every deployment aspect would be handled in this context. This does not seem appropriate because it became apparent that manifest-related model-elements exist that are relevant in entirely different phases of a typical development project.

This aspect is taken as the main motivation that next to the application design it is necessary to subdivide the definition of the term Manifest in three different partitions:

Application Design This kind of description specifies all design-related aspects that apply to the creation of application software for the AUTOSAR AP. It is not necessarily required to be deployed to the adaptive platform machine, but the application design aids the definition of the deployment of application software in the Execution manifest and Service Instance Manifest.

Execution manifest This kind of Manifest is used to specify the deployment-related information of applications running on the AUTOSAR AP.
An Execution manifest is bundled with the actual executable code to support the integration of the executable code onto the machine.

Service Instance Manifest This kind of Manifest is used to specify how service-oriented communication is configured in terms of the requirements of the underlying transport protocols.
A Service Instance Manifest is bundled with the actual executable code that implements the respective usage of service-oriented communication.

Machine Manifest This kind of Manifest is supposed to describe deployment-related content that applies to the configuration of just the underlying machine (i.e. without any applications running on the machine) that runs an AUTOSAR AP.
A Machine Manifest is bundled with the software taken to establish an instance of the AUTOSAR AP.

The temporal division between the definition (and usage) of different kinds of Manifest leads to the conclusion that in most cases different physical files will be used to store the content of the three kinds of Manifest.

In addition to the Application Design and the different kinds of Manifest, the AUTOSAR Methodology supports a **System Design** with the possibility to describe Software Components of both AUTOSAR Platforms that will be used in a System in one single model. The Software Components of the different AUTOSAR platforms may communicate in a service-oriented way with each other. But it is also possible to describe a mapping of Signals to Services to create a bridge between the service-oriented communication and the signal-based communication.

3.5 Application Design

The application design describes all design-related modeling that applies to the creation of application software for the AUTOSAR AP.

The Application Design focuses on the following aspects:

- Data types used to classify information for the software design and implementation
- Service interfaces as the pivotal element for service-oriented communication
- Definition how service-oriented communication is accessible by the application
- Persistency Interfaces as the pivotal element to access persistent data and files
- Definition how persistent storage is accessible by the application
- Definition how files are accessible by the application
- Definition how crypto software is accessible by the application
- Definition how the Platform Health Management is accessible by the application
- Definition how Time Bases are accessible by the application
- Serialization properties to define the characteristics how data is serialized for the transport on the network
- REST service interfaces as the pivotal element to communicate with a web service by means of the REST pattern
- Description of client and server capabilities
- Grouping of applications in order to ease the deployment of software.

The artifacts defined in the application design are independent of a specific deployment of the application software and thus ease the reuse of application implementations for different deployment scenarios.

3.6 Execution manifest

The purpose of the execution manifest is to provide information that is needed for the actual deployment of an application onto the AUTOSAR AP.
The general idea is to keep the application software code as independent as possible from the deployment scenario to increase the odds that the application software can be reused in different deployment scenarios.

With the execution manifest the instantiation of applications is controlled, thus it is possible to

- instantiate the same application software several times on the same machine, or to
- deploy the application software to several machines and instantiate the application software per machine.

The Execution manifest focuses on the following aspects:

- Startup configuration to define how the application instance shall be started. The startup includes the definition of startup options and access roles.
  Each startup may be dependent on machines states and/or function group states.
- Resource Management, in particular resource group assignments.

3.7 Service Instance Manifest

The implementation of service-oriented communication on the network requires configuration which is specific to the used communication technology (e.g. SOME/IP). Since the communication infrastructure shall behave the same on the provider and the requesters of a service, the implementation of the service must be compatible on both sides.

The Service Instance Manifest focuses on the following aspects:

- Service interface deployment to define how a service shall be represented on the specific communication technology.
- Service instance deployment to define for specific provided and required service instances the required credentials for the communication technology.
- Configuration of E2E protection
- Configuration of Security protection
- Configuration of Log and Trace

3.8 Machine Manifest

The machine manifest allows to configure the actual adaptive platform instance running on a specific hardware (machine).

The Machine Manifest focuses on the following aspects:

- Configuration of the network connection and defining the basic credentials for the network technology (e.g. for Ethernet this involves setting of a static IP address or the definition of DHCP).
- Configuration of the service discovery technology (e.g. for SOME/IP this involves the definition of the IP port and IP multicast address to be used).
- Definition of the used machine states
- Definition of the used function groups
- Configuration of the adaptive platform functional cluster implementations (e.g. the operating system provides a list of OS users with specific rights).
- Configuration of the Crypto platform Module
- Configuration of Platform Health Management
- Configuration of Time Synchronization
- Documentation of available hardware resources (e.g. how much RAM is available; how many processor cores are available)

4 Operating System

4.1 Overview

The Operating System (OS) is responsible for run-time scheduling, resource management (including policing memory and time constraints) and inter-process communication for all Applications on the Adaptive Platform. The OS works in conjunction with Execution Management which is responsible for platform initialization and uses the OS to perform the start-up and shut-down of Applications.

The Adaptive Platform does not specify a new Operating System for highly performant processors. Rather, it defines an execution context and Operating System Interface (OSI) for use by Adaptive Applications.

The OSI specification contains application interfaces that are part of ARA, the standard application interface of Adaptive Application. The OS itself may very well provide other interfaces, such as creating processes, that are required by Execution Management to start an Application. However, the interfaces providing such functionality, among others, are not available as part of ARA and it is defined to be platform implementation dependent.

The OSI provides both C and C++ interfaces. In the case of a C program, the application's main source code business logic include C function calls defined in the POSIX standard, namely PSE51 defined in IEEE1003.13 [1]. During compilation, the compiler determines which C library from the platform's operating system provides these C functions and the applications executable shall be linked against at runtime. In case of a C++ program, application software component's source code includes function calls defined in the C++ Standard and its Standard C++ Library.

4.2 POSIX

There are several operating systems on the market, e.g. Linux, that provide POSIX compliant interfaces. However, applications are required to use a more restricted API to the operating systems as compared to the platform services and foundation.

The general assumption is that a user Application shall use PSE51 as OS interface whereas platform Application may use full POSIX. In case more features are needed on application level they will be taken from the POSIX standard and NOT newly specified wherever possible.

The implementation of Adaptive Platform Foundation and Adaptive Platform Services functionality may use further POSIX calls. The use of specific calls will be left open to the implementer and not standardized.

4.3 Scheduling

The operating system provides multi-threading and multi-process support. The standard scheduling policies are SCHED_FIFO and SCHED_RR, which are defined by the POSIX standard. Other scheduling policies such as SCHED_DEADLINE or any other operating system specific policies are allowed, with the limitation that this may not be portable across different AP implementations.

4.4 Memory management

One of the reasons behind the multi-process support is to realize ‘freedom of interferences’ among different Functional Clusters and AA. The multi-process support by OS forces each process to be in an independent address space, separated and protected from other processes. Two instances of the same executable run in different address spaces such that they may share the same entry point address and code as well as data values at startup, however, the data will be in different physical pages in memory.

4.5 Device management

Device management will be provided under POSIX PSE51 interfaces. Refer to POSIX specifications for details.

5 Execution Management

5.1 Overview

Execution Management is responsible for all aspects of system execution management including platform initialization and startup/shutdown of Applications. Execution Management works in conjunction with the Operating System to perform run-time scheduling of Applications.

5.2 System Startup

When the Machine is started, the OS will be initialized first and then Execution Management is launched as one of the OS's initial processes. Other functional clusters and platform-level Applications of the Adaptive Platform Foundation are then launched by Execution Management. After the Adaptive Platform Foundation is up and running, Execution Management continues launching Adaptive Applications. The startup order of the platform-level Applications and the Adaptive Applications are determined by the Execution Management, based on Machine Manifest and Execution manifest information.

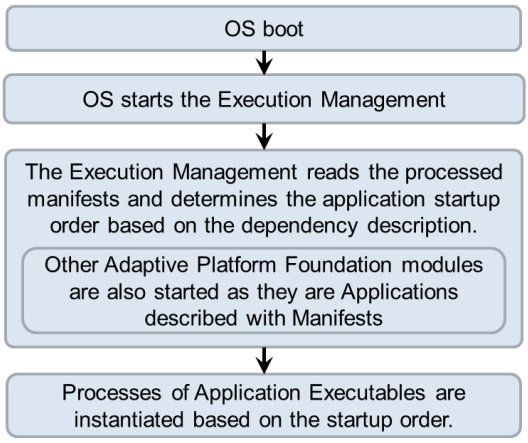

Figure 5-1 AP start-up sequence

5.3 Execution Management Responsibilities

Execution Management is responsible for all aspects of Adaptive Platform execution management and Application execution management including:

1. Platform Lifecycle Management
   Execution Management is launched as part of the Adaptive Platform startup phase and is responsible for the initialization of the Adaptive Platform and deployed Applications.

2. Application Lifecycle Management
   The Execution Management is responsible for the ordered startup and shutdown of the deployed Applications. The Execution Management determines the set of deployed Applications based on information in the Machine Manifest and Execution manifests and derives an ordering for startup/shutdown based on declared Application dependencies. Depending on the Machine State and on the Function Group States, deployed Applications are started during Adaptive Platform startup or later, however it is not expected that all will begin active work immediately since many Applications will provide services to other Applications and therefore wait and “listen” for incoming service requests.

The Execution Management is not responsible for run-time scheduling of Applications since this is the responsibility of the Operating System. However, the Execution Management is responsible for initialization/configuration of the OS to enable it to perform the necessary run-time scheduling based on information extracted by the Execution Management from the Machine Manifest and Execution manifests.

5.4 Deterministic Execution

Deterministic execution provides a mechanism such that a calculation using a given input data set always produces a consistent output within a bounded time. Execution Management distinguishes between time and data determinism. The former states that the output is always produced by the deadline whereas the latter refers to generating the same output from the same input data set and internal state.

The support provided by Execution Management focuses on data determinism as it assumes time determinism has handled by the provision of sufficient resources. For data determinism, Execution Management provides the DeterministicClient APIs to support control of the process-internal cycle, a deterministic worker pool, activation time stamps, and random numbers. In the case of software lockstep, the DeterministicClient interacts with an optional software lockstep framework to ensure identical behavior of the redundantly executed Processes. DeterministicClient interacts with Communication Management to synchronize data handling with cycle activation.

The API supported by DeterministicClient and its interaction with an application is illustrated in Figure 5-2 Deterministic Client

.

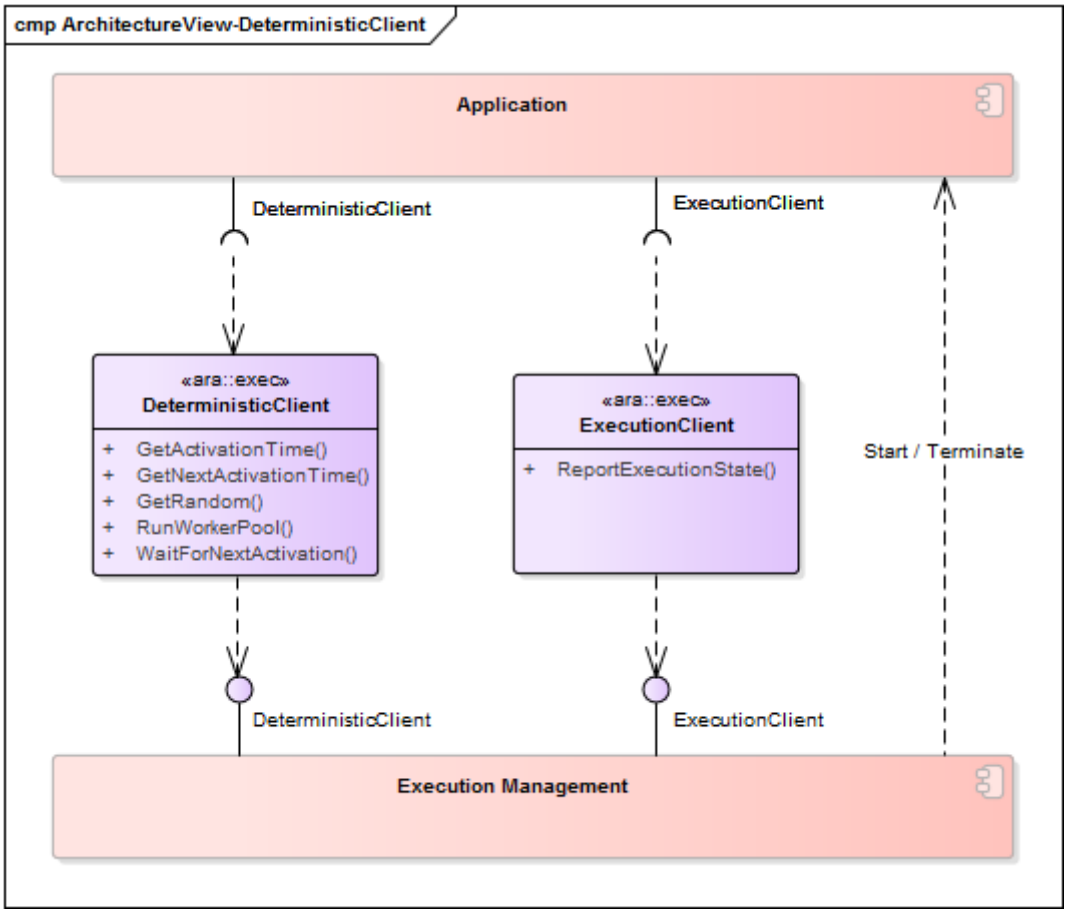

Figure 5-2 Deterministic Client

5.5 Resource Limitation

The Adaptive Platform permits execution of multiple Adaptive Applications on the same Machine and thus ensuring freedom from interference is a system property. Hence an incorrectly behaving Adaptive Application should be limited with respect to its ability to affect other applications, for example, an application should be prevented from consuming more CPU time than specified due to the potential for consequent impacts on the correct functioning of other applications.

Execution Management supports freedom from interference through the configuration of one or more ResourceGroups to which application's processes are assigned. Each ResourceGroup may then be assigned a limit for CPU time or memory that permits restricting the Application's available resources.

5.6 State Management

State Management provides a mechanism to define the state of the operation for an Adaptive Platform. State Management grants full control over the set of Applications

to be executed and ensures that processes are only executed (and hence resources allocated) when needed.

The Machine States and the Function Group States define the current set of running processes. Each process declares in its Execution manifest in which States it shall be active.

Four different states are relevant to Execution Management:

- Machine State
  The Machine States are mainly used to control machine lifecycle (startup/ shutdown/restart), platform level processes, and other infrastructure. There are several mandatory Machine states that must be present on each machine. Additional machine specific Machine States can be defined in the Machine Manifest.
- Function Group State
  Function Group States are mainly used to individually start and stop groups of functionally coherent user level Application processes. They can be configured in the Machine Manifest.
- Process State
  Process States are used for Application Lifecycle Management and are implemented by an Execution Management internal state machine.
- Execution State
  The Execution State characterizes the internal lifecycle of any instance of an Application Executable, i.e. process. Each process must report Execution State changes to Execution Management.

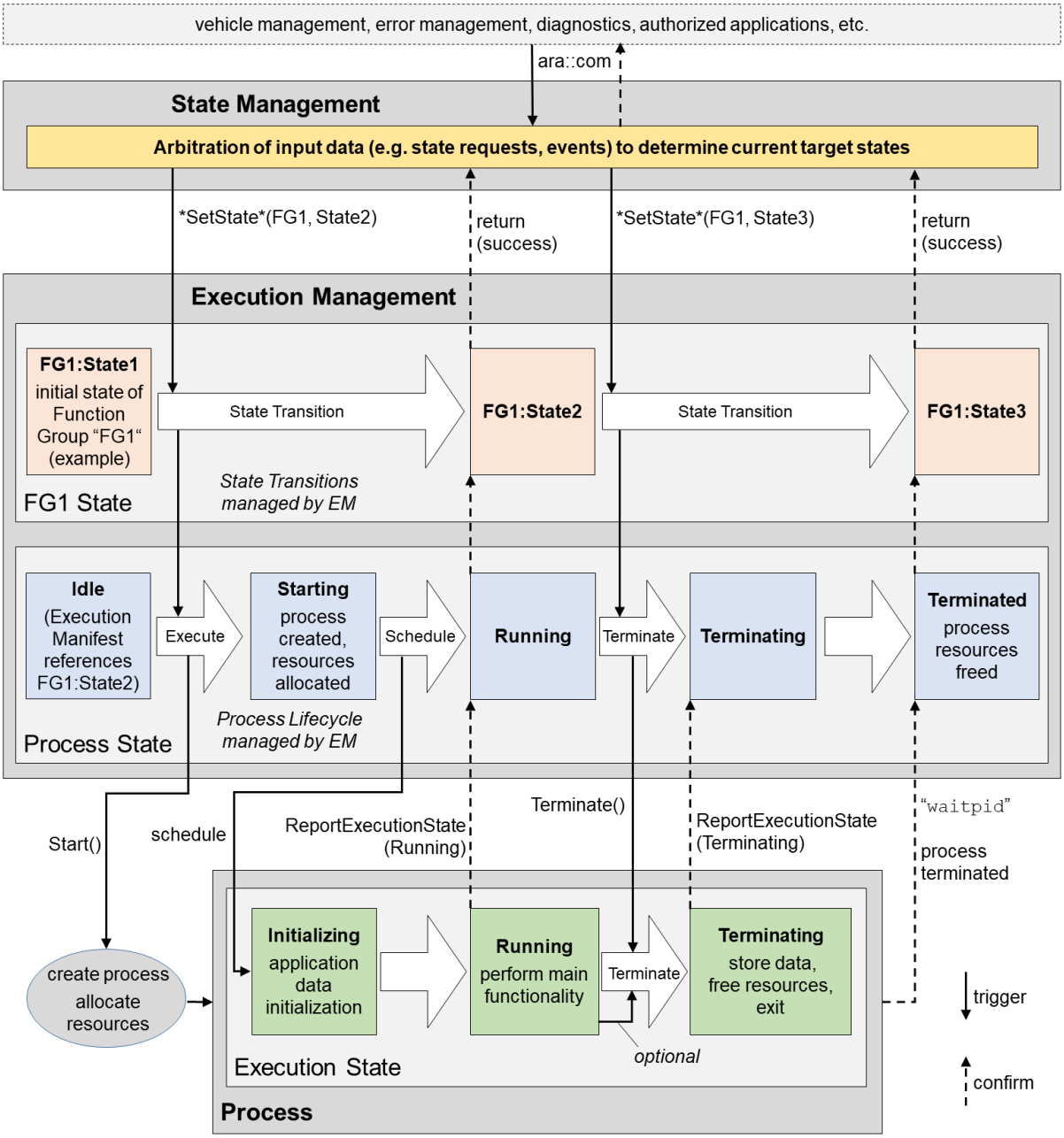

Figure 5-3 Interaction between States

5.7 Application Recovery

Execution Management is responsible for the state-dependent management of Process start/stop, so it has to have the special right to start and stop Processes. The Platform Health Management monitors Processes and could trigger a Recovery Action in case any Process behaves not within the specified parameters. The Recovery Actions are defined by the integrator based on the software architecture requirements for the Platform Health Management and configured in the Execution Manifest.

6 State Management

State Management is the Functional Cluster which is responsible for defining the current set of Machine State and Function Group States, and for initiating State transitions by requesting them from Execution Management. Execution Management performs the State transitions and controls the actual set of running Processes, depending on the current States.

State Management is the central point where new Machine States and the Function Group States can be requested and where the requests are arbitrated, including coordination of contradicting requests from different sources. Additional data and events might need to be considered for arbitration.

The State change requests can be issued by:

- Platform Health Management to trigger error recovery, e.g. to activate fallback functionality
- Diagnostics, to switch the system into diagnostic states
- Update and Config Management to switch the system into states where software or configuration can be updated
- Network Management to coordinate required functionality and network state
- authorized applications, e.g. a vehicle state manager which might be located in a different machine or on a different ECU

State Change requests can be issued by other Functional Clusters via ara::com service interfaces.

Since State Management functionality is critical, access from other Functional Clusters or Applications must be secured, e.g. by IAM (Identity and Access Management). State Management is monitored and supervised by Platform Health Management.

State Management provides interfaces to request information about current states.

Additionally State Management provides functionality to control Processes in a more fine-grained way e.g. to support late ‘wake-up’, to carry out application-specific reset actions or to control communication behavior of an application. All this is done without the need to remove Process from memory and reload/restart them with a different set of startup parameters.

State Management functionality is highly project-specific, and AUTOSAR decided against specifying functionality like the Classic Platforms BswM for the Adaptive Platform for now. It is planned to only specify a set of basic service interfaces, and to encapsulate the actual arbitration logic into project specific code (e.g. a library), which can be plugged into the State Management framework and has standardized interfaces between the framework and arbitration logic, so the code can be reused on different platforms.

The arbitration logic code might be individually developed or (partly) generated, based on standardized configuration parameters. These and other design decisions are still under discussion, and details will be provided at a later point in time.

7 Communication Management

7.1 Overview

The Communication Management is responsible for all aspects of communication between applications in a distributed real-time embedded environment.

The concept behind is to abstract from the actual mechanisms to find and connect communication partners such that implementers of application software can focus on the specific purpose of their application.

7.2 Service Oriented Communication

The notion of a service means functionality provided to applications beyond the functionality already provided by the basic operating software. The Communication Management software provides mechanisms to offer or consume such services for intra-machine communication as well as inter-machine communication.

A service consists of a combination of

- Events
- Methods
- Fields

Communication paths between communication partners can be established at design-, at startup- or at run-time. An important component of that mechanism is the *Service Registry* that acts as a brokering instance and is also part of the Communication Management software.

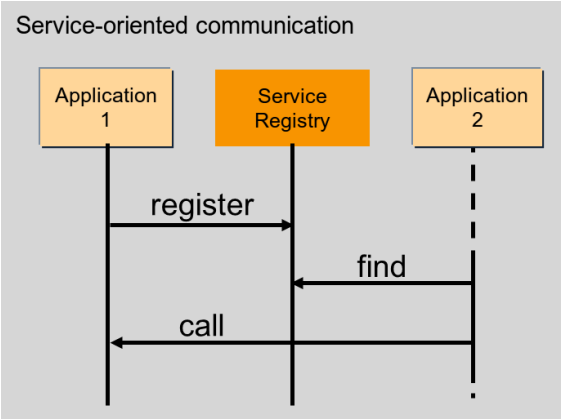

Figure 7-1 Service-oriented communication

Each application that provides services registers these services at the *Service Registry*. To use a service a consuming application needs to find the requested

service by querying the *Service Registry*, this process is known as *Service Discovery*.

7.3 Language binding and Network binding

The Communication Management provides standardized means how a defined service is presented to the application implementer (upper layer, *Language Binding*) as well as the respective representation of the service's data on the network (lower layer, *Network Binding*). This assures portability of source code and compatibility of compiled services across different implementations of the platform.

The *Language binding* defines how the methods, events, and fields of a service are translated into directly accessible identifiers by using convenient features of the targeted programming language. Performance and type safety (as far as supported by the target language) are the primary goals. Therefore, the *Language Binding* is typically implemented by a source code generator that is fed by the service interface definition.

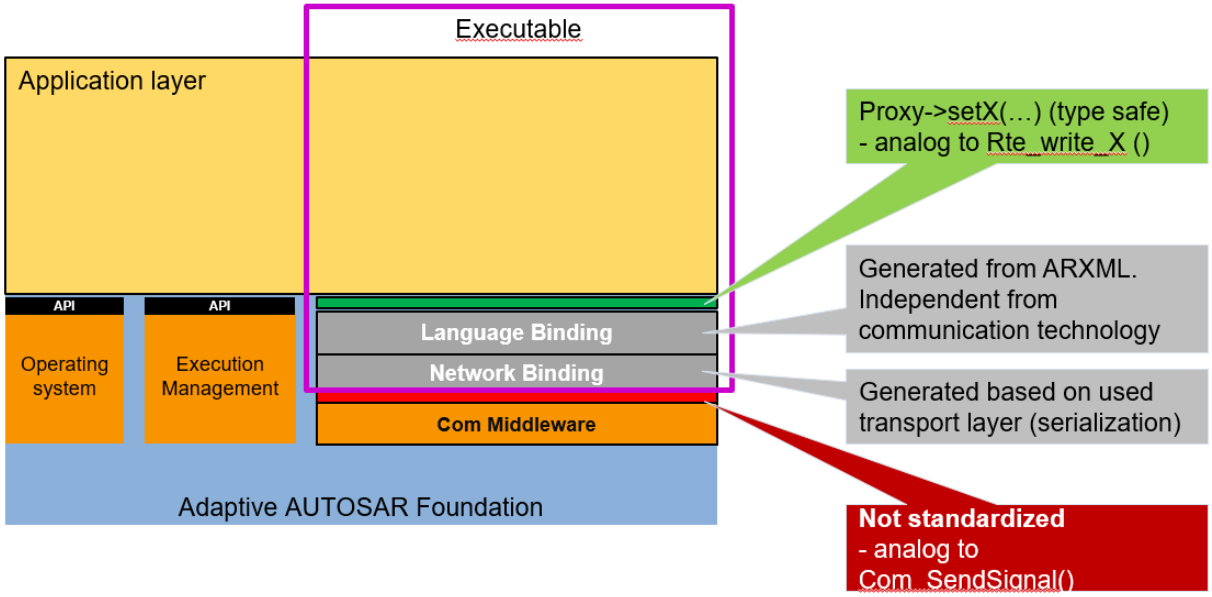

Figure 7-2 Example Language and Network Binding

The *Network Binding* defines how the actual data of a configured service is serialized and bound to a specific network. It can be implemented based on Communication Management configuration (interface definition of the AUTOSAR meta model) either by interpreting a generated service specific recipe or by directly generating the serializing code itself.

The local *Service Registry* is also part of the *Network Binding*.

Please note: the interface between *Language Binding* and *Network Binding* is considered as a private interface inside Communication Management software. Therefore, a normative specification defining this interface is currently out of scope. Nevertheless, platform vendors are encouraged to define independently such an interface for their software to allow for easy implementation of other *Language Bindings* than C++ together with other *Network Bindings* inside their platform implementation.

7.4 Generated Proxies and Skeletons of C++ Language Binding

The upper layer interface of the C++ Language Binding provides an object-oriented mapping of the services defined in the interface description of the AUTOSAR meta model.

A generator that is part of the development tooling for the Communication Management software generates C++ classes that contain type safe representations of the fields, events, and methods of each respective service.

On the service implementation side, these generated classes are named *Service Provider Skeletons.* On the client side, they are called *Service Requester Proxies*.

For Service Methods, a Service Requester Proxy provides mechanisms for synchronous (blocking the caller until the server returns a result) and asynchronous calling (called function returns immediately). A caller can start other activities in parallel and receives the result when the server's return value is available via special features of the Core Type `ara::core::future`. See chapter 18.1.

A platform implementation may be configured such that the generator creates mock-up classes for easy development of client functionality when the respective server is not yet available. The same mechanism can also be used for unit testing the client.

Whereas proxy classes can be used directly by the client the *Service Provider Skeletons* for the C++ binding are just abstract base classes. A service implementation shall derive from the generated base class and implement the respective functionality.

The interfaces of ara::com can also provide proxies and skeletons for safety-related E2E protected communication. These interfaces are designed that compatibility to the applications is assured independent whether E2E protection is switched on or off.

7.5 Static and dynamic configuration

The configuration of communication paths can happen at design-, at startup- or at run-time and is therefore considered either static or dynamic:

- **Full static configuration**:
  service discovery is not needed at all as the server knows all clients and clients know the server.
- **No discovery by application code:**
  the clients know the server but the server does not know the clients. Event subscription is the only dynamic communication pattern in the application.
- **Full service discovery in the application**:
  No communication paths are known at configuration time. An API for Service discovery allows the application code to choose the service instance at runtime.

8 RESTful Communication

8.1 Overview

Both communication stacks, ara::com and ara::rest can establish communication paths between Adaptive Applications. ara::rest is a framework to build RESTful APIs as well as specific services on top of such an API. It does not define a specific API out-of-the-box to construct directly RESTful services. This framework is modular, it enables developers to access different layers involved in RESTful message transactions directly. In contrast, the focus of ara::com is to provide a traditional function call interface and to hide all details of the transactions beyond this point. Another important difference is that ara::rest ensures interoperability with non-AUTOSAR peers. For example, an ara::rest service can communicate with a mobile HTTP/JSON client and vice versa.

8.2 Architecture

The Architecture of ara::rest is based on a modular design which supports developers at the level of API as well as service design. The following diagram illustrates its general design. It depicts how a service application is composed in ara::rest.

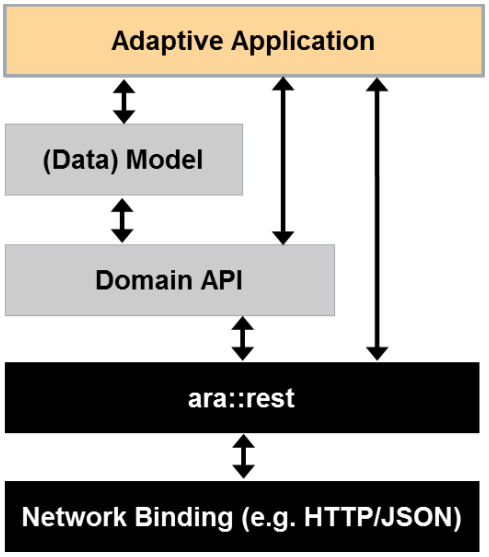

Figure 8-1 ara::rest stack architecture overview

The generic REST layer of ara::rest only provides three fundamental abstractions: A tree-structured message payload (Object Graph), a URI and a request method (like GET or POST known from HTTP). From these basic primitives domain-specific RESTful APIs can be composed which defines a concrete high-level protocol for interaction via object graphs, URI and methods. Its purpose is to define the rules for access into a domain-specific data model and to provide an abstract (C++) API to an application. Instead of using this Domain API, it is also possible for an Adaptive Application to use ara::rest directly when this further abstraction is not needed.

8.3 Components

ara::rest comprises of the following set of components.

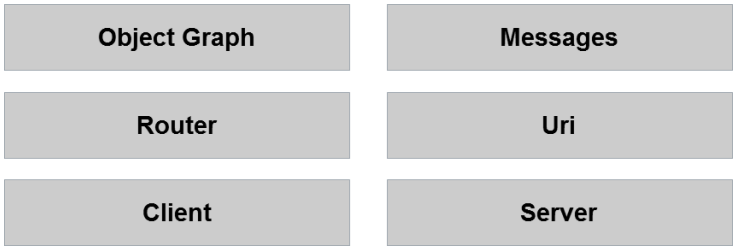

Figure 8-2 ara::rest components

The Object Graph is a protocol-binding independent tree-like data structure which is the cornerstone of all ara::rest communication. Its purpose is to map to a protocol format such as JSON as well as to C structs. This maximizes compatibility with non-ARA communication peers and Classic AUTOSAR. Object graphs are transmitted in messages which abstract completely from a concrete underlying protocol binding. Still they enable a user to access protocol-specific details if required.

Messages encapsulate the entire context of a request/reply communication cycle in the asynchronous programming model of ara::rest.

The routing concept provides a means to map requests (including request method and URI) onto user-defined handler functions. Routing is the cornerstone to lift abstraction from generic REST into a specific kind of RESTful API.

Uri is a generic RFC-compliant but highly efficient URI representation.

ara::rest provides so-called (network) endpoints for server and client communication which both provide a comparable degrees of resource control. Both are designed to provide fast and efficient communication capabilities on single as well as multi-core systems.

The entire framework design is strictly geared towards maximal resource control. All computations and allocations can be strictly controlled and customized to the precise needs of an application (deployment).

9 Diagnostics

9.1 Overview

The Diagnostic Management realizes the ISO 14229-5 (UDSonIP) which is based on the ISO 14229-1 (UDS) and ISO 13400-2 (DoIP).

The Diagnostic Management represents a functional cluster of the Adaptive Platform on the service layer using ara::com. Therefore, it is language independent and may be able to serve Adaptive Applications with other language bindings e.g. Java in the future.
The configuration is based on the AUTOSAR Diagnostic Extract Template (DEXT) of the Classic Platform. DEXT starts to be settled in the market and is already used and supported by several OEMs and vendors.

The supported Transport Layer is DoIP. Future Adaptive Platforms will support further Transport Layers e.g. CAN. Maybe also customized Transport Layers are also planned to be supported because DoIP is typically not used as in-vehicle protocol.

The scope is to abstract the diagnostic protocol from Adaptive Applications. The interfaces are harmonized with the Classic Platform (e.g. SetEventStatus) to allow an easy change for Classic Platform developers.

Traditionally in AUTOSAR Classic Platform, Diagnostics is intended for one physical ECU typically with a microcontroller running the DCM and DEM functions. However, the AUTOSAR Adaptive Platform considers further use-cases of ECUs with more than one processing units (multiple micro-controllers, microprocessors, GPUs and so on), as well as dynamic extensibility with further applications. This will require a new mechanism to address different parts of the AP machine.

The atomic updateable/extendable parts are managed by *SoftwareClusters (SWCL).* A *SoftwareCluster* contains all parts which are relevant to update installed or deploy a particular set of new functionalities/applications. Hence the Adaptive Diagnostics Manager supports an own diagnostics Server instance for each installed *SoftwareCluster* having its own *DiagnosticAddress*. Note that this Software Cluster is also coupled with the Software Package of UCM so that the Software Cluster can be updated or newly introduced to a machine.

9.2 Diagnostic communication sub-cluster

The diagnostic communication sub-cluster is like the DCM of the Classic Platform – it realizes the diagnostic server. Currently, the supported services are limited, but the support of further UDS services will be extended in future releases.

Besides the pseudo-parallel client handling of ISO 14229-1, the Diagnostic Manager (DM) is extended to support a full parallel handling of different diagnostic clients. This satisfies the demands of modern vehicle architectures including several diagnostic clients (tester) for data collection, access from the backend, SOTA (Software Over-the-Air) and finally the classic workshop and production use-cases.

Diagnostic aware Adaptive Application (AA)
In this case, the DM dispatches an incoming diagnostic request (typically routine control or DID related service) to an AA, which provides an explicit diagnosis related interface (service interface specific to UDS service type. E.g. the service interface for a routine control consists of methods "start", "requestResults" and "stop" and each method defines specific UDS error codes as application errors).

Parameters parsed/serialized by AA itself from/to UINT8-Array
In this case, the entire UDS data-parameters starting with data-parameter#1 in the request and the entire UDS data-parameters starting with data-parameter#1 in the positive response are given as IN/OUT parameters as Vectors of type UINT8 to the service method. For such use-cases, where no specific mapping to a UDS request is intended, but simple forwarding, and to have a very flexible way of request handling, the GenericUDSService interface has been introduced.

Parameters are given as typed in/out method parameters
In this case, the entire UDS data-parameters starting with data-parameter#1 in the request and the entire UDS data-parameters starting with data-parameter#1 in the positive response are given as distinct IN/OUT parameters of data type according to the type definition of the DiagnosticDataElement related to the data-parameter#N in DiagExt.

9.3 Event memory sub-cluster

The event memory sub-cluster is like DEM of the Classic Platform – it is responsible for DTC management.

The supported functionality and interface are like the Classic Platform. The diagnostic monitor is represented as (Diagnostic-)Event which can be combined with a DTC. The DTC can be assigned to PrimaryMemory (accessible via 19 02/04/06) or to configurable UserMemories (accessible via 0x19 17/18/19). The DTC can store Snapshot- and ExtendedDataRecords.

Counter- and Timebase Debouncing are supported. Furthermore, DM offers notifications about internal transitions: interested parties are informed about DTC status byte changes, the need of monitor re-initialization for DiagnosticEvents and if the Snapshot- or ExtendedDataRecord is changed.

The operation cycle changes – important for the aging and readiness calculation – need to be forwarded to the DM.

Same applies for the storage- and enable conditions – changes need to be forwarded to DM. By enabling conditions the general update of DTCs can be controlled e.g. to disable all network related monitors within under voltage conditions. By storage conditions, the DTC cannot be stored in the DTC memory.

10 Persistency

10.1 Overview

Persistency offers mechanisms to applications and other functional clusters of the Adaptive Platform to store information in the non-volatile memory of an Adaptive Machine. The data is available over boot and ignition cycles. Persistency offers standard interfaces to access the non-volatile memory.

The Persistency APIs take storage location identifiers as parameters from the application to address different storage locations.

The available storage locations fall into two categories:

- Key-Value Storage
- File-Proxy Storage

Every application may use a combination of multiple of these storage types.

Persistent data is always private to one application. There is no mechanism available to share data between different applications using the Persistency. This decision was taken to prevent a second communication path beneath the functionality provided by Communication Management.

Persistency offers encryption for stored data to make sure that sensitive data will be encrypted before storing it on a physical device.

10.2 Key-Value Storage

The Key-Value Storage provides a mechanism to store and retrieve multiple Key-Value pairs in one storage location. The following three kinds of data types are supported directly by Key -Value Storage:

- Data types defined in SWS_AdaptivePlatformTypes.
- Simple byte arrays that result from a streaming of complex types in the application.
- All Implementation Data Types referred via “dataTypeForSerialization“ by a “PersistencyKeyValueDatabaseInterface” or specialized as PersistencyDataElements of that interface in the Application Design

The keys need to be unique for each Key-Value database and are defined by an application using the methods provided by the Persistency.

Adding serialization/storage support based on application/platform specific serialization code for AUTOSAR data types which are defined in Application Design is planned.

10.3 File-Proxy Storage

Not all data relevant for persistent storage is structured in such a way that Key-Value databases are a suitable storage mechanism.
For this kind of data the mechanism of File-Proxy Storage was introduced.
A File-Proxy Port allows an application to access a storage location and create one or multiple accessors in it. These accessors again are identified by unique keys in string format.
To give a better impression of this mechanism, a comparison to a file system helps: a File-Proxy Port can be understood as a filesystem directory in which an application is allowed to create multiple files (accessors).

Since File-Proxy Storage is close to classical file system access, the API was designed as a subset of the well-known C++ std::iostream class with similar behavior.

10.4 Use cases for handling persistent data for UCM

Handling the persistent data/persistent files of UCM use cases by Persistency during the UCM process purely depends on persistency configuration.

In general, there are three main use cases supported in UCM for handling adaptive applications over the life cycle of the CAR ECU or adaptive machine.

- Installation of new application software to the Adaptive Machine
- Update of existing application software to the Adaptive Machine
- Uninstallation of the existing application software from the Adaptive Machine

In all three scenarios, Persistency is used by UCM to deploy/delete/update the persistent data of an application.

Persistency shall support the below-mentioned scenarios.

- Persistency shall be able to deploy the persistent data to a Key-Value database or File-Proxy that was defined by an application designer during the Adaptive Application installation

- Persistency shall be able to deploy the persistent data to Key-Value database or File-Proxy that was changed by an integrator

- Persistency shall be able to deploy the persistent data to Key-Value database or File-Proxy that was defined by an integrator

- Persistency shall be able to overwrite or retain the persistent data to Key-Value database or File-Proxy as per the update strategies configured for the

Key-Value database or File-Proxy when a new version of an application is installed

- Persistency shall be able to remove the persistent data Key-Value database or File-Proxy when an application is uninstalled

In general, the Persistency layer is configured during application design and deployment. Persistency shall be able to use the deployment stage configuration to override the application design configuration. If deployment stage configurations are missing then configuration from the application design will be considered for the deployment of persistent data.

Persistency shall check the newly installed and updated persistent data before integration into a Key-Value database or a File-Proxy

11 Time Synchronization

11.1 Overview

Time Synchronization (TS) between different applications and/or ECUs is of paramount importance when the correlation of different events across a distributed system is needed, either to be able to track such events in time or to trigger them at an accurate point in time.

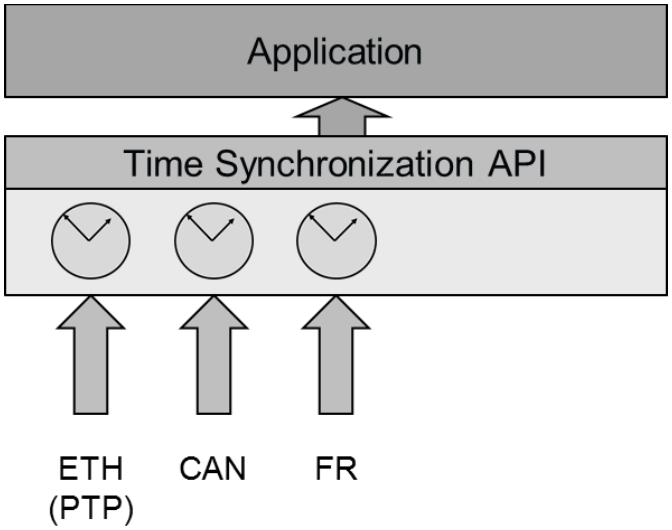

Figure 11-1 Time Synchronization

For this reason, a Time Synchronization API is offered to the Application, so it can retrieve the time information - synchronized with other Entities / ECUs.

The Time Synchronization functionality is offered by the means of different "Time Base Resources" which are present in the system.

11.2 Design

For the Adaptive Platform, the following three different technologies were considered to fulfill all necessary Time Synchronization requirements:

- StbM of the Classic Platform
- Library chrono - either std::chrono (C++11) or boost::chrono
- The Time POSIX interface

The TimeSynchronization module provides a similar functionality like the StbM module of the Classic Platform but with a std::chrono inspired API design.

The following functional aspects are considered by the Time Synchronization module:

- Startup Behavior
- Constructor Behavior (Initialization)
- Normal Operation
- Error Handling

The following functional aspects will be considered in future releases:

- Shutdown Behavior
- Error Classification
- Version Check

11.3 Architecture

The application will have access to a different specialized class implementation for each Time Base Resource (TBR).

The TBRs are offered as a Resource in a similar way as Services are offered in the ara::com design and therefore it is adopting the following architectural design patterns of ara::com:

- Proxy: Similar to the ara::com Service Proxy Skeleton pattern, TS provides a Resource Proxy pattern, omitting the Skeleton part.

- Find: Similar to the ara::com Service Proxy Find the pattern, TS provides a Resource Proxy Find pattern to provide access to TBRs.

- Proxy Methods: Similar to the ara::com Proxy Methods pattern, TS uses a Methods pattern also adhering to the asynchronous Future pattern.

This architectural design puts the Time Synchronization design apparently in a frontal conflict when talking about avoiding latencies, since the latter are inherently added by the asynchronous behavior of the design pattern of the ara::com API.

12 Network Management

12.1 Overview

NOTE: The Network Management (NM) is intended to be controlled via State Management as the control of partial network needs to be coordinated with the set of the relevant application via Function Group State of EM controlled by SM. The contents in this chapter do not yet reflect the design.

The AUTOSAR NM is based on a decentralized network management strategy, which means that every network node performs activities independently depending only on the NM packets received and/or transmitted within the communication system.

The AUTOSAR NM algorithm is based on periodic NM packets, which are received by all nodes in the cluster via multicast messages.
The reception of NM packets indicates that sending nodes want to keep the NM-cluster awake. If any node is ready to go to sleep mode, it stops sending NM packets, but as long as NM packets from other nodes are received, it postpones the transition to sleep mode. Finally, if a dedicated timer elapses because no NM packets are received any more, every node performs the transition to the sleep mode.
If any node in the NM-cluster requires bus-communication, it can keep the NM-cluster awake by starting the transmission NM packets.

12.2 Architecture

The Adaptive Platform specification describes the functionality, the API design and the configuration of the Network Management for the AUTOSAR Adaptive Platform independently of the underlying communication media used. At the moment only Ethernet is considered but the architecture is kept bus – independent.

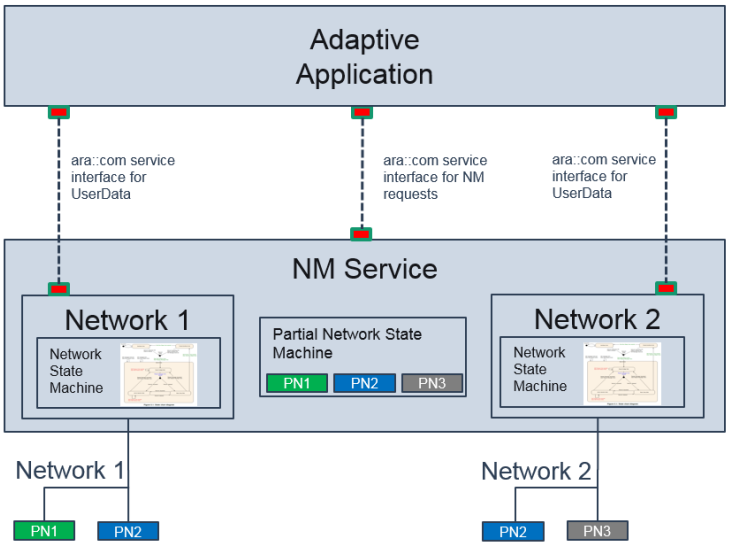

Figure 12-1 Overview NM

Its main purpose is to coordinate the transition between normal operation and bus-sleep mode of the underlying logical networks (a logical network can be either a physical or a partial network) in internally coordinated state machines.

It provides Service Interfaces to Adaptive Applications for requesting and releasing networks. It coordinates the request of various applications and provides an aggregated machine request over the network.
In addition to the core functionality, services are provided to the applications for accessing the user data information in the NM message.

13 Update and Config Management

13.1 Overview

One of the declared goals of Adaptive AUTOSAR is the ability to flexibly update the software and its configuration through over-the-air updates. To support changes in the software on an Adaptive Platform, the Update and Configuration Manager(UCM) provides an Adaptive Platform service that handles software update requests.

UCM is responsible for updating, installing, removing and keeping a record of the software on an Adaptive Platform. Its role is similar to known package management systems like dpkg or YUM in Linux, with additional functionality to ensure a safe and secure way to update or modify software on the Adaptive Platform.

13.2 Update protocol

UCM service has been designed to support the diagnostic use case of Diagnostic communication sub-software cluster and support performing changes in the Adaptive Platform in safe, secure and resource efficient update process. To fulfill requirements to support updates of several clients and to enable fast download, UCM has to transfer Software Packages (UCM input) separately from their processing.

13.2.1 Data transfer

Data transfer is done by streaming data over ara::com. This enables transferring data into UCM without the need to buffer data outside UCM. UCM stores packages into a local repository where packages can be processed in the order requested by the client of UCM.

As the transfer phase is separated from the processing phase, UCM supports receiving data from multiple clients without limitations when this can be done.

13.3 Packages

13.3.1 Software package

The unit for installation which is the input for the UCM is a Software Package. Package includes, for example, one or several executables of (Adaptive) Applications, operating system or firmware updates, or updated configuration and calibration data that shall be deployed on the Adaptive Platform. This constitutes the Updateable Package part in Software Packages and contains the actual data to be added or changed for the Adaptive Platform. Beside application and configuration data, each Software Package contains an SW Package manifest providing metadata

like package name, version, dependencies and possible some vendor specific information for processing the package.

The format of the Software Package isn't specified, which enables using a different kind of solutions for the implementationon of UCM. Software Package consists of updates to be performed in software and metadata. This content is pushed through vendor tooling to generate Software Package which will be processed by the UCM in the target. For the deployment of the Software Package into the target, OEM can wrap Software Package into OEM deployment container. In target, OEM specific application will receive the OEM package and perform possible uncompressing and decryption of the package before passing it to UCM.

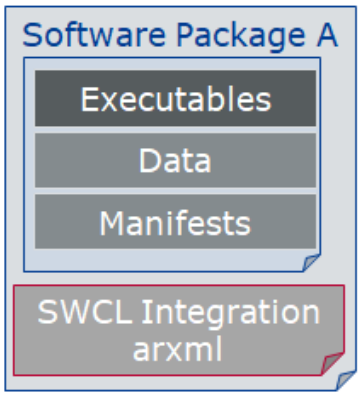

Figure 13-1 Overview Software Package

UCM processes the vendor-specific Software Package based on the provided metadata.

13.3.2 Backend package

In order for an OEM backend to understand packages contents from several package suppliers, a backend package format is specified as described in below picture:

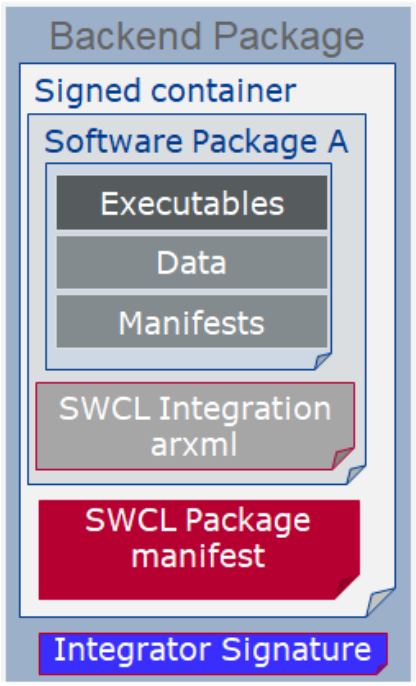

Figure 13-2 Overview Backend Package

The software package format is vendor specific. However, as backend package is meant to be vendor independent, SWCL package manifest must use ARXML file format.

You can find below for information purpose a description of the fields that must be contained in SWCL Package manifest:

General information

- Package name: fully qualified short-name.
- Version: Version for SWCL that has to follow https://semver.org semantic versioning specification with the exception build number is mandatory for debugging/tracking purpose.
- Previous version: update can only be applied if the previous version is present, otherwise can be empty or null
- Package type: boolean, activated if the content has to be processed for delta package.
- Dependencies: Manifest Specification document contained model has to be followed. It describes the dependencies after SWCL is updated or installed. Below is a model example:

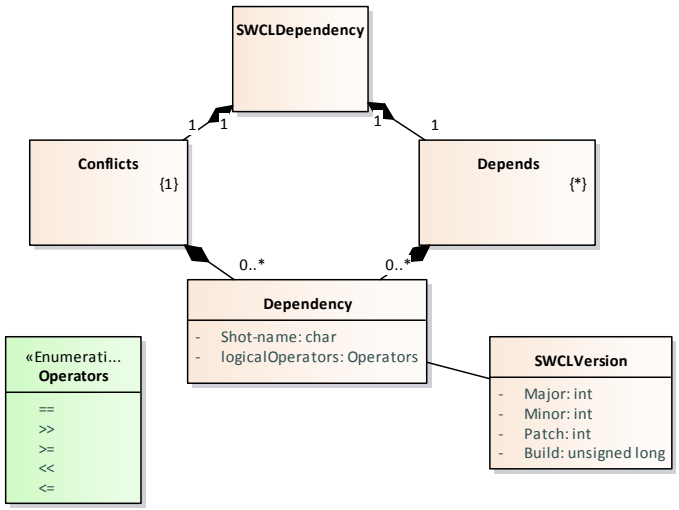

Figure 13-3 Dependency model example

Sizes to allow if there is enough memory available

- Size of SWCL package
- SWCL payload size: size of uncompressed package content, the SWCL size

For information and tracking purpose

- Vendor: vendor id
- Vendor signature
- Packager: vendor id
- Package signature: for package consistency check and security purposes. This can be used to sign the software package in a backend package but also the package containing the data, executables, etc. inside an SWCL integration ARXML.
- Description
- Changes: description of this release change
- License: for instance MIT, GPL, BSD, proprietary.

To distribute the package to the correct UCM:

- Diag address: in case package is coming from the tester via UDS
- UCM identifier: unique identifier within vehicle architecture

For package distribution within the vehicle

- Package payload content type: can be Operating System, configuration, persistent data or executable
- Action: optional, can be updated, removeSWCL or install

13.3.3 Vehicle Package Manifest

A vehicle package manifest is typically assembled by an OEM backend. It contains a collection of software cluster package manifests extracted from backend packages stored in the backend database. It also contains a Vehicle Package manifest that is a

merge of the related backend packages SWCL package manifests along with a campaign orchestration and other fields needed for packages distribution within the vehicle.

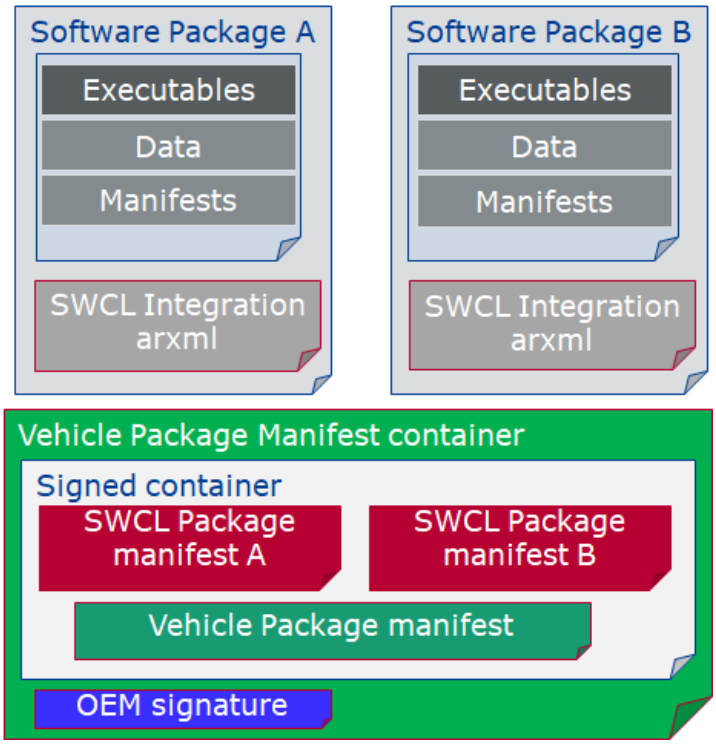

Figure 13-4 Overview Vehicle Package

You can find below for information purpose a description of the fields that must be contained in Vehicle Package manifest:

- List of backend packages: a list of SWCL names
- Dependencies: dependencies between SWCLs that will overrule the already defined dependencies in SWCL Package Manifest. Typically used by vehicle systems integrator to add dependencies related to vehicle systems that backend package supplier is not aware of.
- Origin: repository or diagnostic address, for history, tracking or security purposes
- Version
- Vehicle target: vehicle description
- Campaign orchestration: Below is a model example.

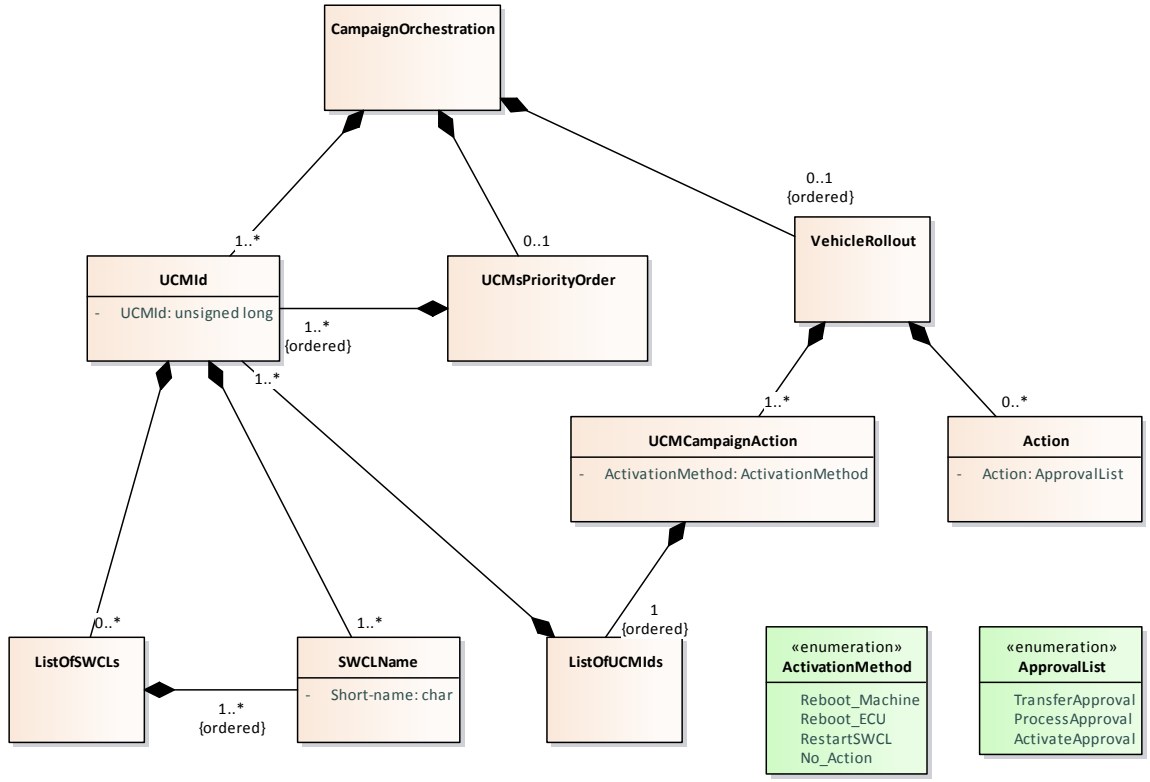

Figure 13-5 Campaign orchestration example model

Vehicle package could be used by a garage to fix a car having issues downloading an update for instance. Therefore, like backend package, Vehicle package manifest should be ARXML file format for interoperability purpose.

13.3.4 Software release and packaging workflow

In order to create a backend package, an integrator has to use a packager compatible with the targeted UCM. This package could be provided by the Adaptive platform stack vendor including the targeted UCM. After integrator is assembling executable, manifests, persistency, etc., he uses a packager to create a software package using UCM vendor specific format. This same software package is then embedded into a backend package along with ARXML SWCL package manifest. The software package could be signed by Adaptive stack vendor packager or integrator and signature included into SWCL package manifest. As backend package might transit via the internet between an integrator and an OEM backend, both software package and SWCL package manifest should be signed into a container along with its signature in order to avoid any SWCL package manifest modification.

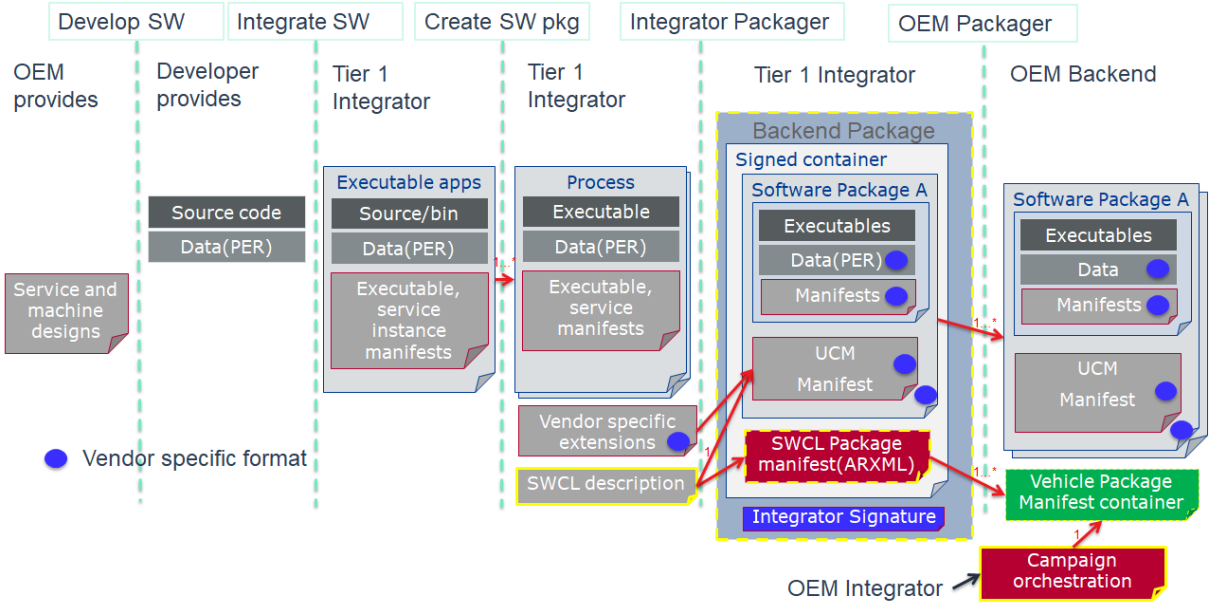

Figure 13-6 Packaging steps

The backend packages assembled by integrator can then be put in the backend database. When a vehicle needs an update or new installation, the backend server will query software packages from backend package and merge the related SWCL package manifests into a vehicle package manifest. In this manifest, backend server embeds a campaign orchestration selected based on vehicle electronic architecture, deducted for instance from Vehicle Identifying Number.

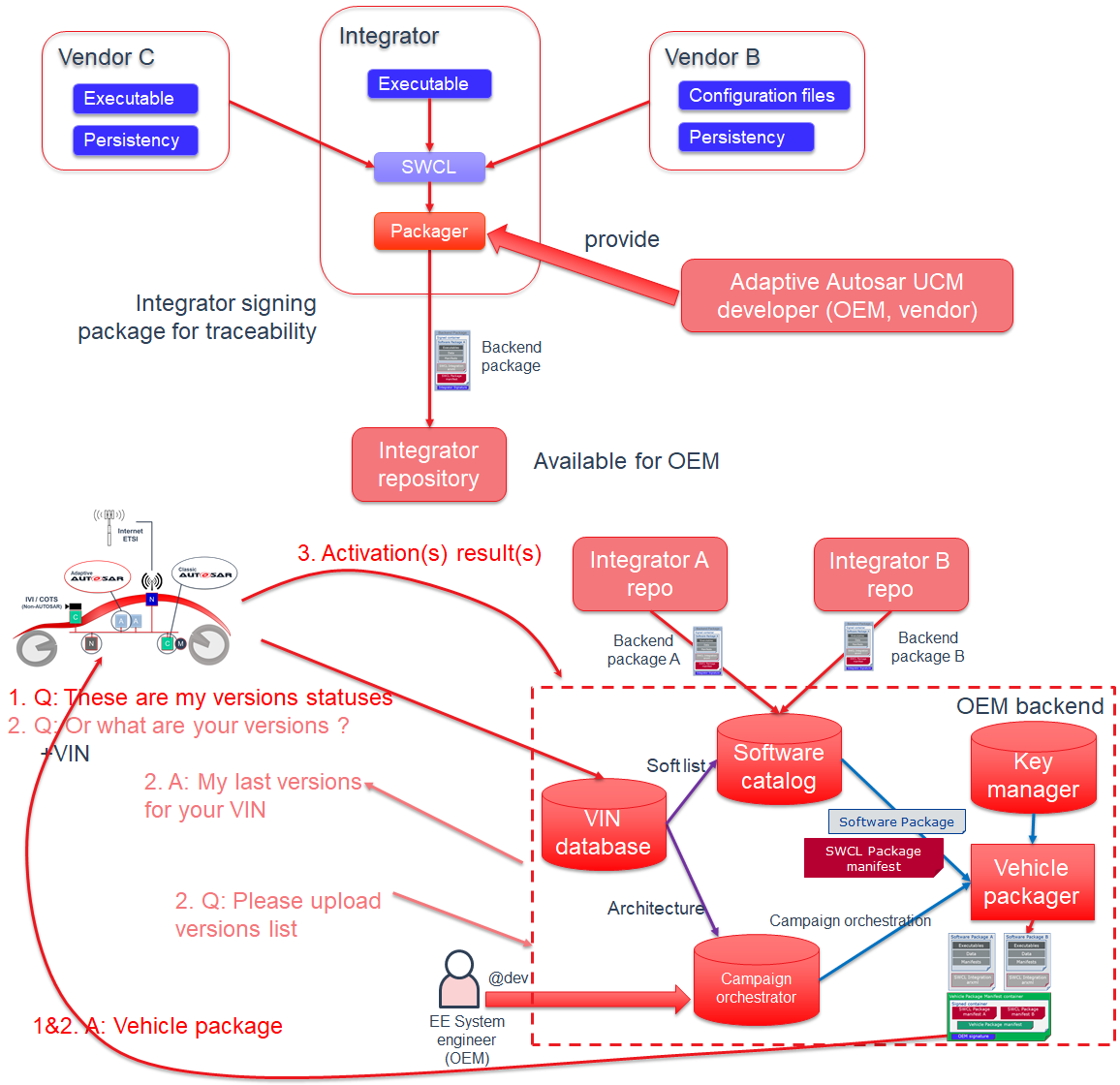

Figure 13-7 Packages distribution to vehicle

13.3.5 Processing and activating Software Packages

Install, update, and uninstall actions are performed through the ProcessSwPackage interface where UCM is able to parse from metadata which actions need to be performed.

UCM sequence has been designed to support for example A/B update scenario or ‘in-place’ scenario (for instance using OSTree) where package manager provides the possibility to rollback into the previous version if this is needed.

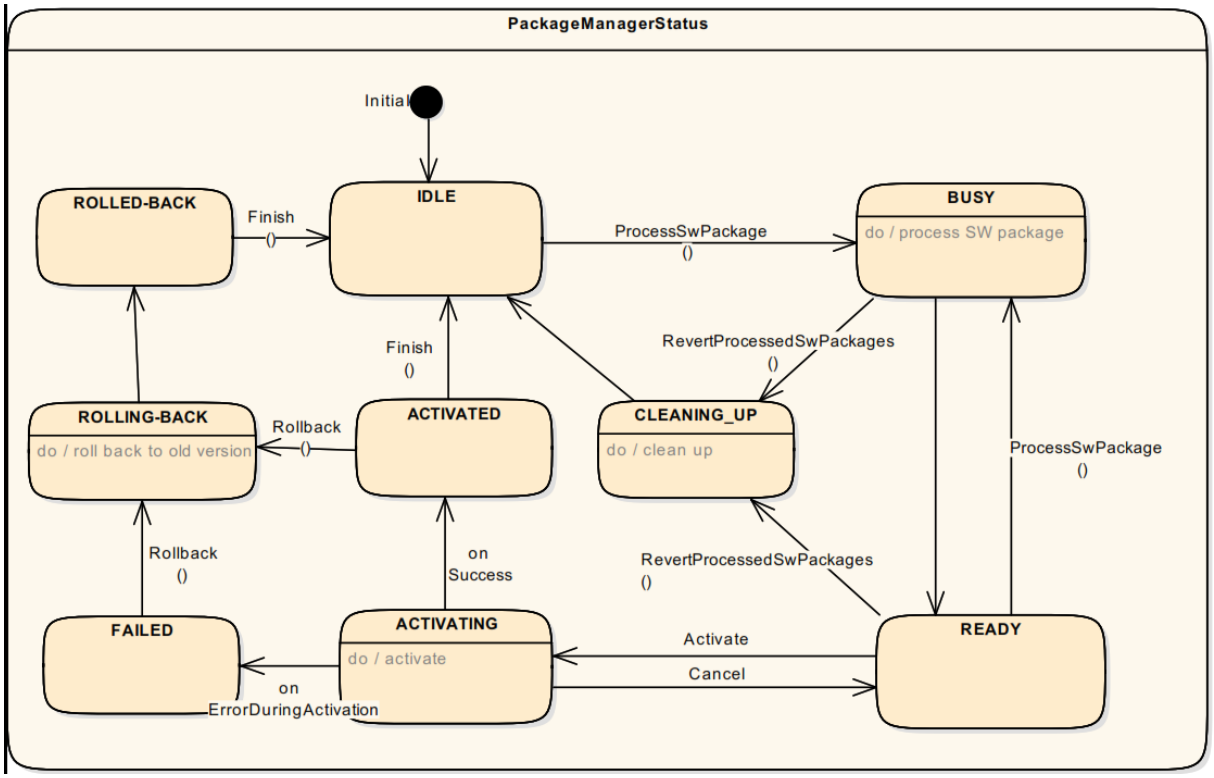

Figure 13-8 Overview Processing and activation Software Package

To keep implementation simpler and more robust, only one client at the time can request to process a Software Package with the ProcessSwPackage method, switching UCM sate to BUSY. Several clients can request to process transferred packages in sequence. In case of A/B partition update scenario, several clients can process the inactive /B partition being updated; in case of software cluster cross dependencies, each client must update in sequence into “B partition”. Once, processing is finished, UCM state switches to READY for an activation or another processing.

Activation of changes with the Activate method is done for all processed packages despite which client requests this. To coordinate this correctly in the multi-client scenario, a “master”-client approach could be used. UCM might not know if all targeted Software Packages have been processed, but it shall perform a dependency check to see that system is consistent with the requirements of the installed software in “B partition”. In case of dependencies aren’t fulfilled UCM shall reject the activation.

When updates are being activated, UCM state changes to ACTIVATING. UCM then performs either a machine reset or a functional group restart depending on the type of update. For instance, if the update includes the operating system of functional cluster update, UCM might want to reset the machine. However, if the update is only about a low criticality function, only restarting functional group could be enough, reducing annoyance to the driver. Once these restarts are finished, UCM switches to ACTIVATED state.

When updates have been ACTIVATED, other processing requests will be rejected until activation has been resolved. In this phase, UCM can verify from PHM and EM if targeted software clusters are running properly. After verification, UCM can either call Finish for success or Rollback for failure.

After Finish is called, UCM cleans all unneeded resources and returns to IDLE.
In case of Rollback is called an A/B partition scenario, UCM reactivates “A partition” and switched to REVERTING state.
In case of Rollback is called an ‘in-place’ update, clients can either try to resolve detected issues and try Activate again or clean up by calling RevertProcessedSwPackages.

13.4 Software information reporting

UCM provides service interfaces that expose functionality to retrieve Adaptive Platform software information, such as names and versions of transferred packages, for processed but not committed software and for the last committed software. As UCM update process has clear states, UCM provides information in which state is the processing of each Software Package.

13.5 Software update consistency and authentication

UCM shall authenticate the Software Package using signature covering the whole Software Package as described in Figure 13-1 Overview Software Package. The Adaptive platform shall provide necessary checksum algorithms, cryptographic signatures or other vendor and/or OEM specific mechanisms to validate the package, otherwise, an error will be returned by UCM. Practically, Software Package should be packaged by the tool coming from the same vendor as the one developing the targeted UCM in order to have authentication algorithm compatibility.

As authentication algorithms are using hashes, consistency is also checked when authenticating a software package. Therefore, once a software package is finished transferring, its authentication and consistency are checked at TransferExit call, before any software package is processed.

13.6 Securing the update process

UCM provides a service over ara::com. There is no authentication step in UCM update protocol. Instead, it is up to Identity and Access Management concept to ensure that the client requesting UCM services over ara::com is legit.

UCM shall not allow updating an older version of software cluster than the one present at processing time in the Adaptive Platform (otherwise an attacker could update to an old package with known security flaw).

13.7 Safe State Management during an update process

Definition of a safe state with respect to the system setup is the OEM responsibility. Based on the system setup and the application, the system might need to be switched into an 'update state', so that they are ignoring missing or faulty messages during the update process.

Additionally, there must be also a minimal check of the system after the update. For this, the OEM specific Diagnostic Application will put the machine into a 'verification state' and check if all the relevant processes have reached the runningState. This gives a chance to perform a Rollback if some processes fail to reach the runningState. Fig. 13-9 provides an overview of this concept.

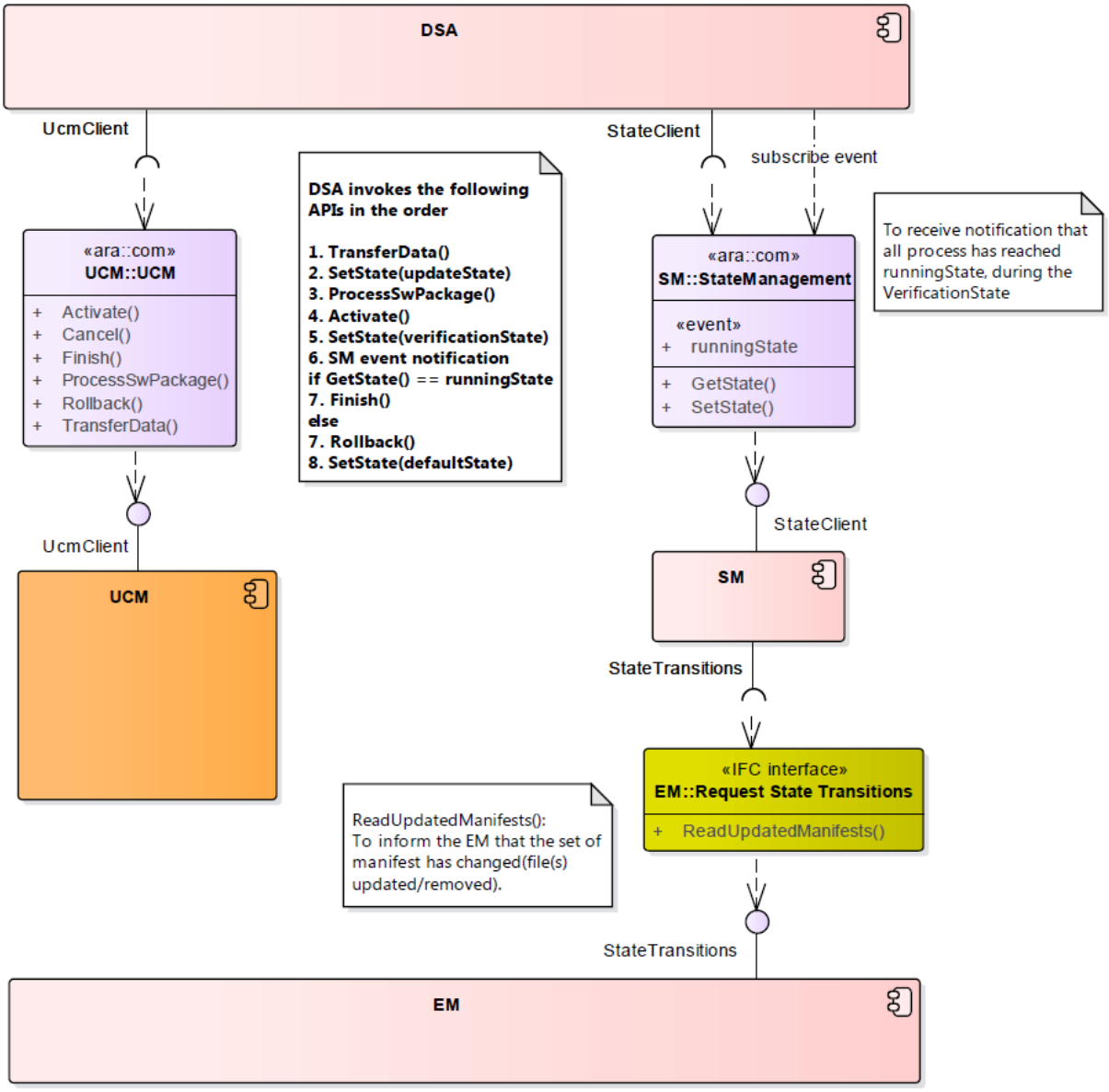

Figure 13-9 State Management during an update process

14 Identity and Access Management

The concept of Identity and Access Management (IAM) is driven by the increasing need for security, as the AUTOSAR Adaptive Platform needs a robust and well-defined trust relationship with its applications. IAM introduces privilege separation for Adaptive Applications and protection against privilege escalation in case of attacks. In addition, IAM enables integrators to verify access on resources requested by Adaptive Applications in advance during deployment. Identity and Access Management provides a framework for access control for requests from Adaptive Applications on Service Interfaces, Functional Clusters of the Adaptive Platform Foundation and related modeled resources.

14.1 Terminology

To understand how the framework works, a few important notions must be defined in advance. As a reference see also 'Terminology for Policy-Based Management' in RFC3198 (https://tools.ietf.org/html/rfc3198).

- **Access Control Decision:** The access control decision is a Boolean value indicating if the requested operation is permitted or not. It is based on the identity of the caller and the Access Control Policy.
- **Access Control Policy:** Access Control Policies are used to define constraints that have to be met in order to access specific objects (e.g. Service Interfaces).
- **Policy Decision Point (PDP):** A PDP makes the access control decision. It determines if an Adaptive Application is allowed to perform the requested task by checking the Access Control Policy.
- **Policy Enforcement Point (PEP):** A PEP interrupts the control flow during requests from Adaptive Applications by requesting the Access Control Decision from a PDP and enforces this decision.
- **Capability:** A capability is a property of an Adaptive Application Identity. Access to an AUTOSAR resource (e.g. Service interface) is granted only if a requesting AA possesses all Capabilities that are mandatory for that specific resource. Capabilities are assigned to AAs within their Application Manifest.
- **Intermediate Identifier (IntID):** An identifier that enables the identification of running POSIX-processes and the mapping to modeled AUTOSAR Processes. The concrete nature of IntID depends on the mechanism that is used to authenticate running POSIX processes.
- **Adaptive Application Identity (AAID):** The Executable that is referenced in a modelled AUTOSAR Process. Multiple AUTOSAR Process that are spawned from the same executable share the same AAID.
- **Adaptive Application Identifier:** A referrer to AAID, i.e. AUTOSAR Process, pointing to exactly one AAID.

14.2 Scope and Focus of the IAM framework:

The IAM framework provides a mechanism for developers of Adaptive AUTOSAR stacks and Adaptive Applications to model the capabilities of each application, to provide access control decisions upon access requests and to enforce the access control. IAM focuses on providing means to limit access from Adaptive Applications to interfaces of the Adaptive Platform Foundation, Service Interfaces and well-defined resources related to Function Clusters (e.g. KeySlots). In particular enforcing quotas on system resources like CPU or RAM is not covered by IAM.

During runtime, the process of IAM is transparent to Adaptive Applications unless a request gets rejected and a notification is raised.

Requests to Service Instances provided by remote Adaptive Platform Instances are covered IAM. PDPs for incoming requests have to be implemented by Adaptive Applications.

The framework is designed to enforce access control to AUTOSAR resources at runtime. It is assumed that Adaptive Applications will be authenticated during startup and that an existing protected runtime environment ensures that Adaptive Applications are properly isolated and prevented from escalating their privileges (i.e., by-passing access control).

14.3 Contents of the AUTOSAR specification

The following table represents which parts of the IAM framework will be defined by AUTOSAR and which parts are up to the developer implementation-wise.

Description	Affiliated to	Part of
Requirement specification for IAM	AUTOSAR Specification	RS_IdentityAndAccessManagement
Behavioral description of the IAM framework (regarding interfaces)	AUTOSAR specification	SWS_IdentityAndAccessManagement
API for communication between AAs implementing a PDP and the PEP in the Adaptive Platform	AUTOSAR Specification	SWS_IdentityAndAccessManagement
API for communication between Functional clusters implementing a PDP and the PEP in the Adaptive Platform.	Not specified by AUTOSAR	-
Application capabilities & Access control policies (Manifest file information).	AUTOSAR specification	TPS_Manifest_Specification

Format and contents of warnings/error messages that the applications receive on failed authorization.	AUTOSAR specification	SWS_IdentityAndAccessManagement
API for activity logging.	AUTOSAR Specification	Not yet decided
Contents of the logging information.	AUTOSAR Specification	Not yet decided
Interface between Adaptive Application and Functional Clusters	Not specified by AUTOSAR	-
Identification of Adaptive Applications during runtime	Not specified by AUTOSAR	-

14.4 The architecture of the IAM Framework

14.4.1.1 General Framework

The IAM architecture divides the authorizing entities logically into an entity that **decides** whether an Adaptive Application is allowed to access a resource (PDP) and an entity that **enforces** the access control decision (PEP). Functional Clusters that need to restrict access to their application interfaces need to implement the PEP that enforces the access control decision provided by a PDP. For that, the PEP will communicate with the PDP if an Adaptive Application requests access to such an interface. Access control decisions are sent back to the PEP based on the request and the applications' capabilities. The necessary information for the access control decision is based on the capabilities found in the Application Manifest of the Adaptive Application that initiated the request as well as the policies. Policies represent the rules that apply for the interfaces, i.e. the preliminaries that an Adaptive Application has to fulfill in order to gather access. Policies are derived from the definition of capabilities and the deployment of resources, e.g. the binding of KeySlots to PortPrototypes bound to Key-Identifier used in Adaptive Applications.

14.4.1.2 Preliminaries and Assumptions

- Applications are designed/configured to have capabilities (properties that allow them to access certain resources)
- Deployed applications will be cryptographically signed to make verification of authenticity possible
- Applications are deployed together with an Application Manifest containing capabilities
- An Adaptive Applications that is subject to IAM has to be started authentically in order and it's manifest has to be authenticated during deployment. The PEP interprets the request and demands a Policy Decision from a PDP (may be implemented in the same process).

14.4.1.3 Identification of Adaptive Applications

In order to request a Policy Decision from a PDP, the PEP has to determine the identity of calling Adaptive Applications. Since every call is mediated through Inter-Process-Communication, the middleware shall support this identification.

The identity itself is a reference to a modeled AA. Capabilities are bound to PortPrototypes and therefore to SWComponentType (see Manifest Specification).

The IAM Framework does not fully specify the identification of AAs. The most appropriate solution heavily depends on the operating system and platform a stack vendor chooses. Many modern operating systems do support the identification of peers on communication endpoints (see SO_PEERCRED in Linux, getpeerid() or Message Passing in QNX). On platforms that do not provide such mechanisms, it might be appropriate to implement a protocol on message-level.

Since Execution Management creates running instances of Adaptive Applications by modeled AUTOSAR Processes it is responsible for keeping track of properties of running processes (i.e. PID of running Adaptive Application) or for assigning properties like setting dedicated UID or assigning keys or UUIDs for message-level implementations. Execution Management shall enable PEPs to find the modeled Adaptive Application for each valid request to the PEP.

The PEP shall be implemented in the Adaptive Foundation and shall be properly isolated from the calling Adaptive Application. PDP shall not be provided by an Adaptive Applications that itself is subject to access control regarding requested action.

14.4.1.4 The IAM Sequence

1. Adaptive Application (AA) initiates request to resource (e.g. Service Interface).
2. PEP interrupts control flow.
3. PEP resolves identity of the requesting Process via EM.
4. PEP passes identity of caller and request parameters to PD.
5. PDP checks if capabilities of AA are sufficient and returns the Access Control Decisions to PEP.
6. PEP enforces Access Control Decision by blocking or allowing the request.

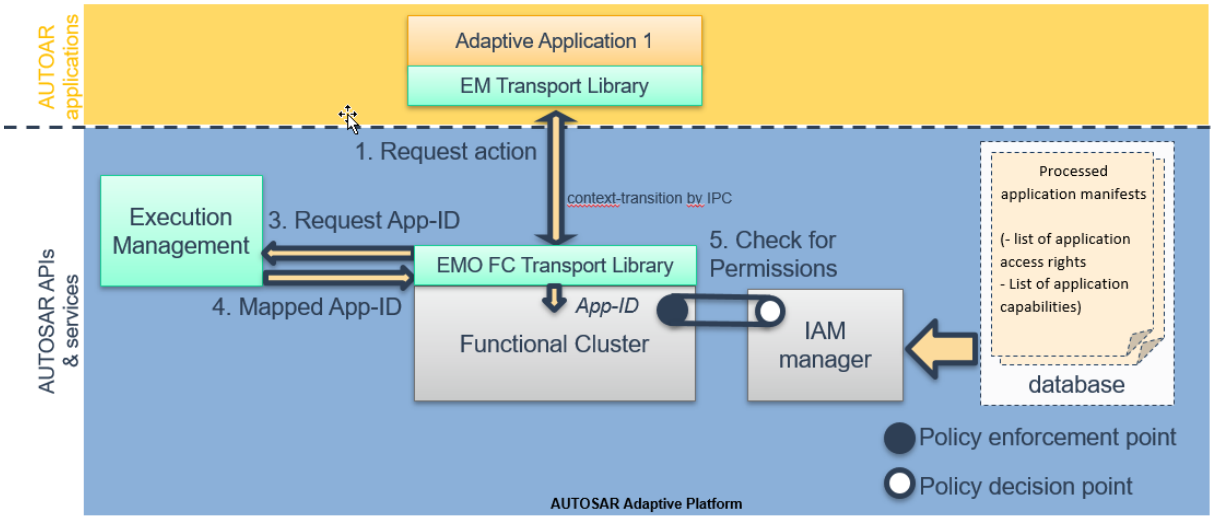

Figure 13-9 IAM Sequence

The Transport Library is aligned with the mechanism used by EM to identify AAs. Giving the example using POSIX-Process-IDs EM tracks the PID retrieved from the operating system during the call to fork(). EM provides this information to PEPs by a protected Functional-Cluster Interface. When using UID EM shall actively set the UID of new POSIX-processes.

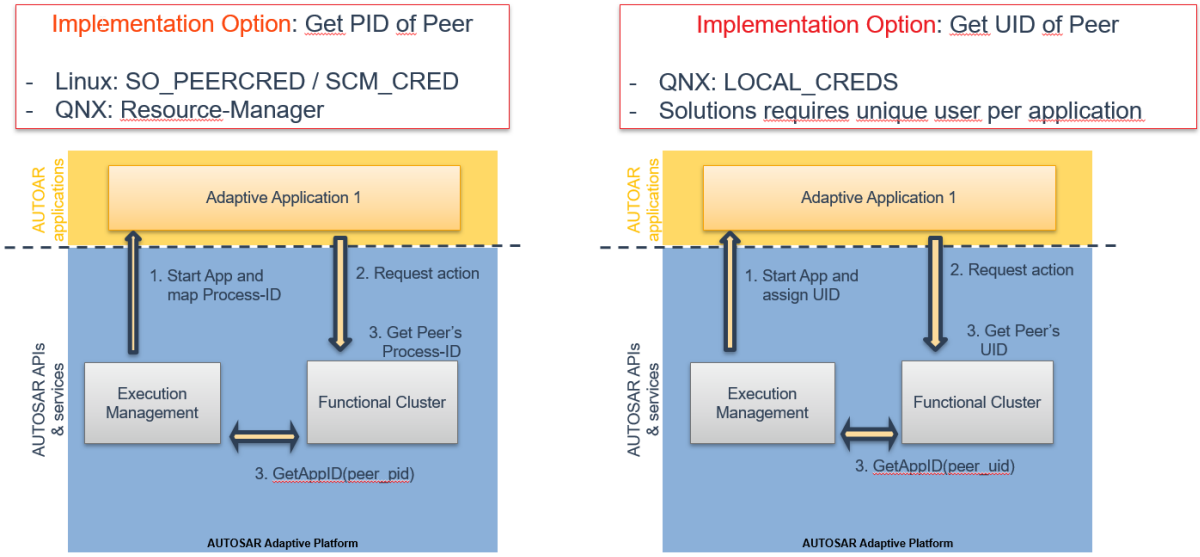

Figure 13-10: Identification of Adaptive Application during runtime, two examples

14.4.1.5 Implementation of Policy Decision Points

Policy Decision Points provide an interface to processed manifest by binary Policy Decisions. Those decisions are based on well-defined Capabilities of Adaptive Applications with their Application Manifest.

Capabilities are modelled by PortPrototypes with semantics specific to single Functional Clusters. Therefore PDPs have to provide interfaces specific to those Capabilities. It is not recommended and not supported by the IAM Framework to address single methods of Functional Clusters by Capabilities. Instead Capabilities are resource-centric. A PDP provides Policy Decisions by checking AAs reference to requested resources.

An example for Capabilities is given by the Crypto API. By assigning the Capability *KeyOwner* referring to a specific modelled key, modifying operations on that key will be allowed. The PDP provides

Application Scenarios in which trusted Adaptive Application implements a PDP are possible and will be specified in the SWS_IdentityAndAccessManagement. Additional information about use-cases regarding this scenario will be provided for AP18-10.

14.5 Inter Platform Communication

As applications may be distributed on different ECUs or virtual machines, we have to deal with identity and access management across platform instances. The following figure shows an example.

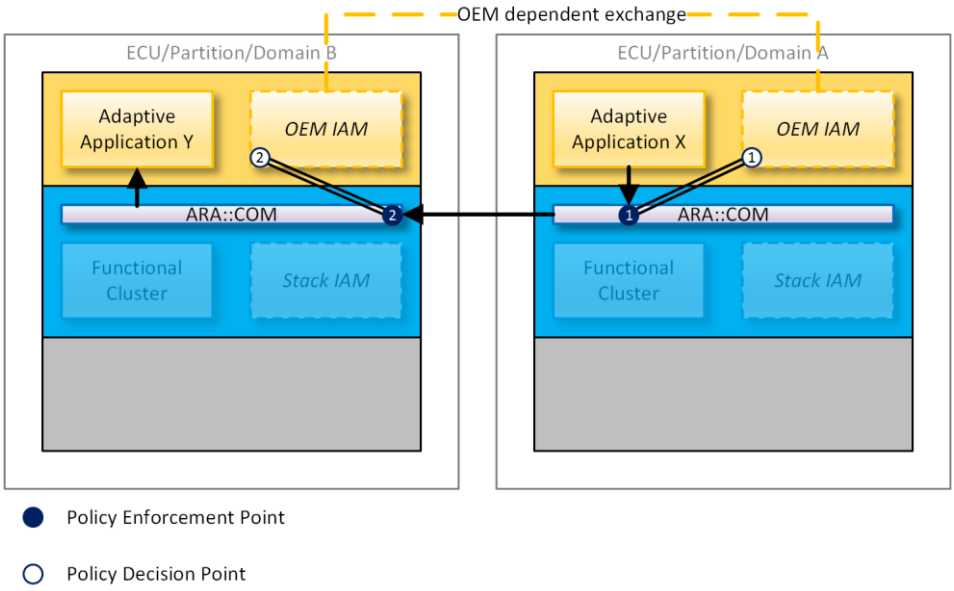

Figure 13-11 IAM Policy Enforcement across Platform Instances

As described above the right-hand platform instance A implements a PEP to check the access rights of application X. We assume here that the PDP is implemented in an OEM-specific Adaptive Application. On the left-hand platform instance B we expect the implementation of a second PEP which is also connected to an OEM-specific PDP. This second PEP is necessary in case A was compromised and does not enforce policies reliably anymore. The second PEP then checks, whether at least one application of A is allowed to access Adaptive Application Y on B. If there is not any capability known to B, access will not be granted.

The set of all capabilities of one instance needs to be synchronized with other instances, such that the second PEP can provide correct enforcement. Since there is no standardized exchange format for the Adaptive Platform we advise the use of an OEM-specific format for capability description. This allows the OEM to define its own synchronization protocol even with non-AUTOSAR platforms.

14.6 Implementation and Usage of IAM

The following list represents the necessary steps (in order) for an FC implementer and system designer to make use of IAM. Further information can be seen in the AUTOSAR_EXP_FCDesignIdentityAndAccessManagement.pdf [4]. Please be aware that the information depicted in the before mentioned document are taken from the AUTOSAR Demonstrator and should only be seen as an implementation proposal/example.

Preparation steps (at design time):

- Applications are designed/configured to have capabilities (properties that allow them to access certain resources) and to see only specific service interfaces

- Deployed applications will be cryptographically signed to make verification of authenticity possible
- Applications are deployed with an execution manifest containing capabilities
- Execution manifest files also contain information like Application ID, how many instances of an application will be instantiated as well as those Application Instance IDs
- The FC implements the enforcement logic referred to as Policy Enforcement Point (PEP)
- The FC is deployed with Policies describing what capabilities are needed to access the provided service interfaces

Usage instructions (at runtime):

- During startup of the Adaptive Platform; EMO will provide a lookup table between Application (Instance) IDs and process IDs
- When an Adaptive Application requests access to a service for which access control is configured, it needs to be authenticated to make referring to its capabilities possible
- The PEP queries the request to the process that implements the PDP (could be the same process)
- The PDP then checks the query for Application ID and its corresponding capabilities and compares it to the stored policies of the FC
- The PDP answers the PEP by sending an access control decision (yes/no)
- The PEP enforces the access control decision itself (granting access based on the decision)

To summarize the steps mentioned, the following should be considered at least:

The FC implementer needs to:

- Provide the following rules which will be put in the service manifest:
  1. What capabilities are needed to access certain services (single or combination of multiple capabilities)
- Implement the logic to query the process that implements the PDP
- Implement the logic to enforce the access control decision that is received

The Application developer needs to:

- Configure capabilities that allow access to the service

15 Cryptography

AUTOSAR Adaptive Platform supports an API for common cryptographic operations and secure key management. The API supports the dynamic generation of keys and crypto jobs at runtime, as well as operating on data streams. To reduce storage requirements, keys may be stored internally in the crypto backend or externally and imported on demand.

The API is designed to support encapsulation of security-sensitive operations and decisions in a separate component, such as a Hardware Security Module (HSM). Additional protection of keys and key usage can be provided by constraining keys to particular usages (e.g., decrypt-only), or limiting the availability of keys to individual applications as reported by IAM.

Depending on application support, the API can also be used to protect session keys and intermediate secrets when processing cryptographic protocols such as TLS and SecOC.

Security Architecture

While AUTOSAR AP only defines the high-level Crypto Stack API exposed to applications, this API is defined with a security architecture in mind that was designed to meet above security and functional requirements.

The general architecture is depicted in Figure 15-1. On the highest layer, AUTOSAR AP, as well as native and hybrid applications, link against the AUTOSAR AP Crypto Stack API. The API implementation may refer to a central unit (Crypto Service Manager) to implement platform-level tasks such as access control and certificate storage consistently across applications. The implementation may also use the Crypto Service Manager to coordinate the offloading of functionality to a Crypto Driver, such as a Hardware Security Module (HSM). Indeed, the offloading functionality of the Crypto Stack API this way is expected to be a typical implementation strategy: The Crypto Driver may implement the complete set of key management and crypto functions in order to accelerate crypto operations and shield managed keys from malicious applications.

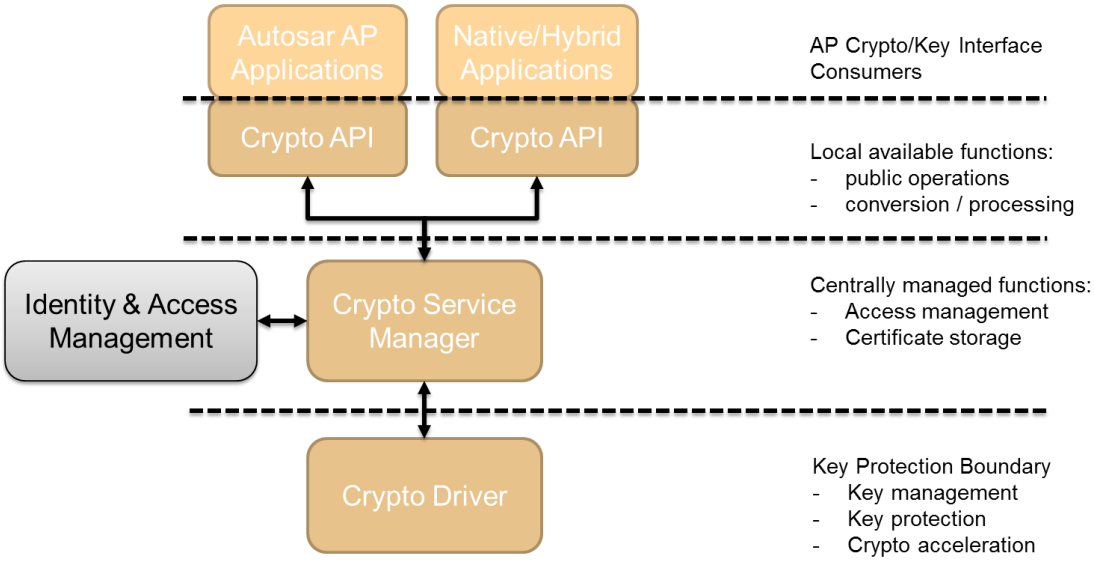

Figure 15-1 Crypto Stack – Reference Architecture

In order to realize this layered security architecture, the Crypto Stack API not only performs bulk crypto operations but also provides native support for:

(1) Operating with encrypted keys or key handles
(2) Managing keys securely despite possible application compromise
(3) Constraining application access to and allowed operations on keys

Key Management Architecture

To support the secure remote management of keys despite potential application compromise, the Crypto Stack integrates a key management architecture where keys and associated data are managed in end-to-end protected form. Keys can be introduced into the system either in a trusted fashion, based on an existing provisioning key, or in an untrusted fashion via local key generation. Assuming an appropriately secured crypto backend/driver, applications are unable to modify keys except via well-defined, authorized requests such as key update or revocation.

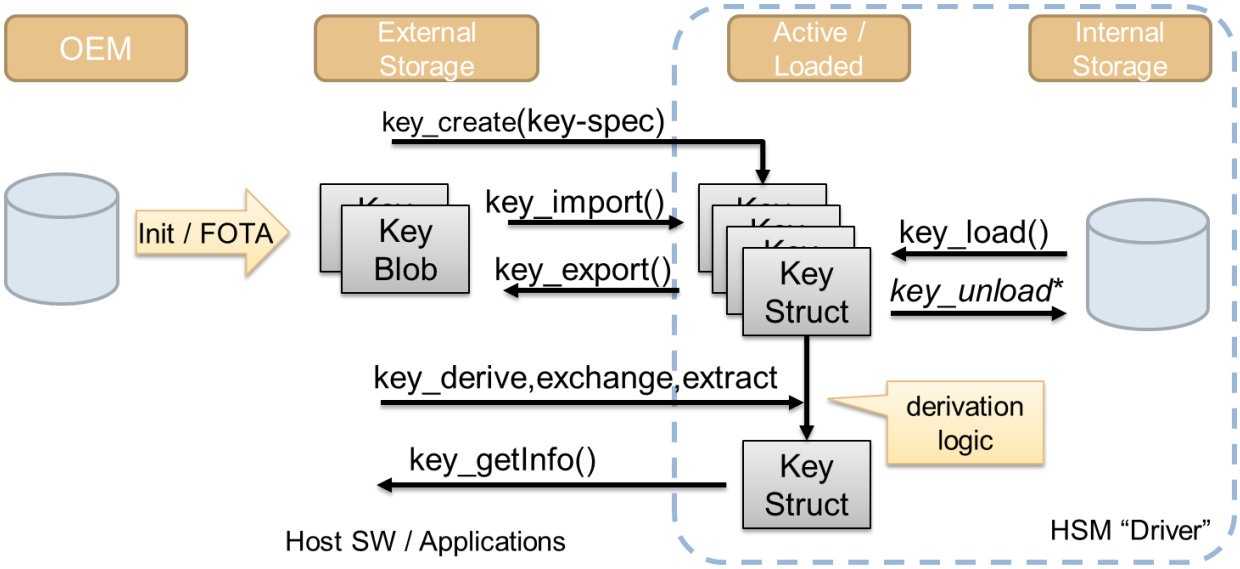

Figure 15-2 CKI Key Management Interactions

Remarks on API Extension

Significant new usages and interactions that require the introduction of new or modified permission/policy validation logic should be tied to corresponding new key usage policy flags. For example, alternative provisioning keys with different ownership/permission checks can be introduced by adding a corresponding new key usage policy and enforcing the new logic in all key management operations involving those new keys.

16 Log and Trace

16.1 Overview

The Log and Trace Functional Cluster is responsible for managing and instrumenting the logging features of the AUTOSAR Adaptive Platform. The logging and tracing features can be used by the platform during development as well as in production. These two use cases differ and the Log and Trace component allows flexible instrumentation and configuration of logging in order to cover the full spectrum. The logging information can be forwarded to multiple sinks, depending on the configuration, such as the communication bus, a file on the system and a serial console. The provided logging information is marked with severity levels and the Log and Trace component can be instrumented to log the information only above a certain severity level, this enables complex filtering and straightforward fault detection of issues on the logging client side. The AUTOSAR Adaptive Platform and the logging Functional Cluster are responsible to maintain the platform stability to not overload the system resources.

The Log and Trace rely on the LT protocol standardized within the AUTOSAR consortium. The protocol ensures that the logging information is packed into a standardized delivery and presentation format. Furthermore, the LT protocol can add additional information to the logging messages, such as an ECU ID. This information can be used by a logging client to relate, sort or filter the received logging frames.

In addition, utility methods are provided, e.g. to convert decimal values into the hexadecimal numeral system or into the binary numeral system. These are necessary to enable applications to provide data to Log and Trace which conforms to the standardized serialization format of the LT protocol.

16.2 Architecture

The Log and Trace interfaces are provided in the namespace ara::log for applications to forward logging onto one of the aforementioned logging sinks.

The Log and Trace interfaces rely on the back-end implementation that is a part of the Logging framework. The Logging framework can use other Functional Clusters to fulfill certain features, such as Time Synchronization or Communication Management.

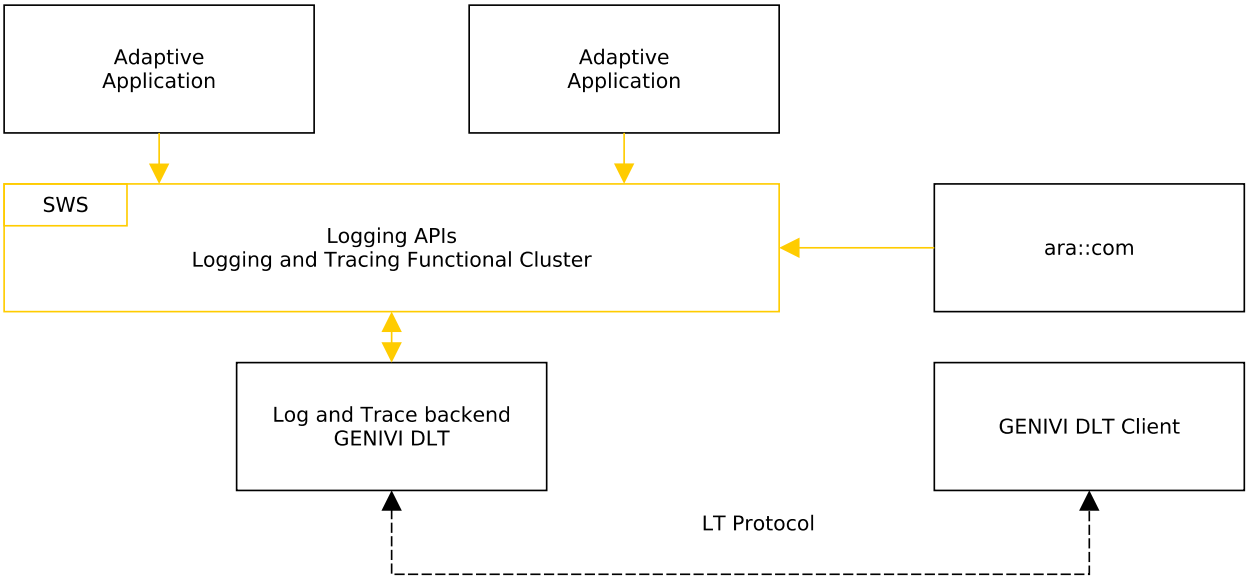

Figure 16-1 Overview Log and Trace

17 Safety

17.1 Safety Overview

AUTOSAR provides a safety overview of the Adaptive Platform to support the integration of the Adaptive Platform in safety projects. For this release in the form of an explanatory document (AUTOSAR_EXP_SafetyOverview).

This document shall help the functional safety engineer to identify functional safety-related topics within the AUTOSAR Adaptive Platform. The content of this document is currently structured into separate chapters as follows and can be mapped on the contents and structures according to ISO 26262:

- AUTOSAR Adaptive Platform objectives, use-cases, and scenarios
- System definition, system context, and assumptions
- Hazard analysis
- Safety Goals
- Functional safety concept and functional safety requirements

The objective of this safety overview is to derive safety requirements from the top level safety goals and assumed use-cases or scenarios and allocate them to the architectural elements of the item, or to an external measure. The use of the AUTOSAR Adaptive Platform does not imply ISO 26262 compliance. It is still possible to build unsafe systems using the AUTOSAR Adaptive Platform safety measures and mechanisms. The architecture of the AUTOSAR Adaptive Platform can, in the best case, only be considered to be a SEooC.

The explanatory document contains assumptions, exemplary items, like reference models, use-cases, scenarios, and/or references to exemplary technical solutions, devices, processes or software. Any such assumptions or exemplary items contained in this document are for illustration purposes only. These assumptions are not part of the AUTOSAR standard.

The chapter Functional safety concept and initial functional safety requirements is still in development and open for discussion and should not be considered mature final.

The chapters

- Technical safety concept and technical safety requirements
- Validation of safety requirements, safety analysis, and exemplary use-cases are scheduled for later releases.

17.2 Protection of information exchange (E2E-Protection)

E2E profiles within AUTOSAR will be supported to allow safe communication between all combinations of AUTOSAR AP and CP instances, whether they are in the same or different ECUs. Where useful, mechanisms will be provided to allow safe communication using more capabilities of the service-oriented approached within the Adaptive Platform. The provided functionality gives the possibility to verify that information sent from a publisher and received by a subscriber has not changed during the transmission. Acknowledgment of transmission and transmission security is not provided in the E2E context according to the E2E mechanism in AUTOSAR CP.

When E2E protection is used in communication between a publisher and a subscriber the E2E protection is invoked synchronously in the process of the publisher. On the subscriber side, the E2E check is invoked at the reception of the data within the subscriber process.

For this release E2E supports:

- Periodic and mixed periodic events in polling mode

The principle for E2E protection of periodic events is that the publisher of an event serializes the event data and adds an E2E header. When receiving the event the subscriber will de-serialize the message and run E2Echeck which will return a result showing if any of the detectable faults (defined by the E2E profiles) occurred during transmission.

Not yet supported are:

- Events in callout mode
- Non-periodic events
- Methods

The profiles that can be used for E2E protection are described in the Autosar Foundation (AUTOSAR_PRS_E2EProtocol)

17.3 Platform Health Management

The Platform Health Management supervises the execution of software. It offers the following supervision functionalities (all supervision functions can be invoked independently):

- Alive supervision
- Deadline supervision
- Logical supervision
- Health Channel Supervision

Alive Supervision checks that a Supervised Entity is not running too frequently and not too rarely.

Deadline supervision checks that step in a Supervised Entity are executed in a time that is within the configured minimum and maximum limits.

Logical supervision checks that the control flow during execution matches the designed control flow.

Health channel supervision provides the possibility to hook external supervision results (like RAM test, voltage monitoring, …) to the Platform Health Management.

Platform Health Management could trigger a configurable recovery action if a failure is detected in the supervised entities.

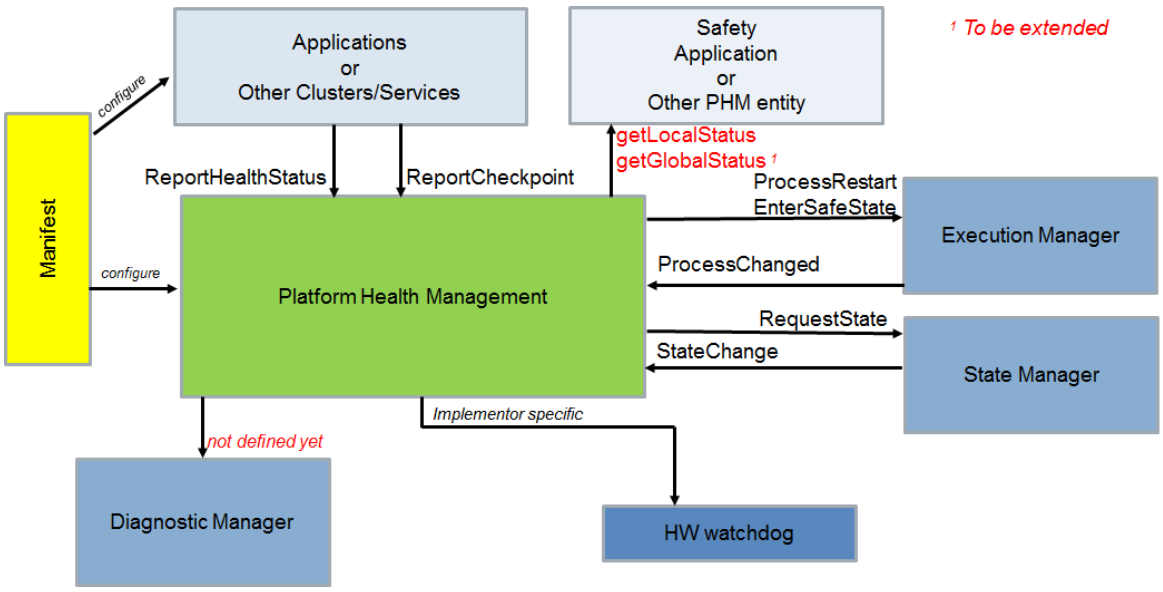

Figure 17-1 Platform Health Management and other functional clusters

The following recovery actions are available for an Autosar Adaptive Platform:

- Request the State Manager to switch to a specified Machine, FunctionGroup or Application state (RequestState API).
- Request the Execution Manager to force switching to a specified Machine or FunctionGroup State (EnterSafeState API). This action shall be configured instead of the corresponding API with the State Manager if the State Manager has issues detected by the supervision mechanisms.
- Request the Execution Manager to restart a specified process (ProcessRestart API).
- Request the Watchdog driver to perform a watchdog reset (implementor specific API).
- Report error information to the Diagnostic Manager: not specified in this release.
- Forward error information to another PHM entity or an Application: not specified in this release.

Known limitations for this release:

- Only a single PHM instance is currently supported. Multiple PHM instances and daisy-chaining of multiple instances is currently not supported.

- Dependency on the Diagnostic Manager is not defined, yet

- Health Management configuration related to Supervision Modes is not fully supported in this release

Functionality shared by CP and AP is described in the foundation documents and named “Health Monitoring” (RS_HealthMonitoring, SWS_HealthMonitoring). Additional specifications for AP only are described in the AP documents and named “Platform Health Management” (RS_PlatformHealthManagement, SWS_PlatformHealthManagement).

17.4 C++ coding guidelines

The main application sector that AUTOSAR C++14 Coding Guidelines document applies to is automotive, but it can be used in other embedded applications that work in a safety-related and critical environment. The AUTOSAR C++14 Coding Guidelines rule set is applicable to high-end embedded micro-controllers that provide efficient and full C++14 language support, on 32 and 64-bit microcontrollers, using POSIX or similar operating systems.

Existing standards are incomplete, covering old C++ versions or not applicable to critical/safety-related. In particular, MISRA C++:2008 does not cover C++11/14. Multiple new language features required analysis of how useful they can be in

providing efficient implementations and how much risk is associated with the usage of each feature.

Several other coding standards and resources are referenced in this document or used as a basis of the rules in this document, like JSF, HIC, CERT and C++ Core Guideline. Most of the rules are automatically enforceable by static analysis that can be applied without a manual code review. Style of code in a sense of naming conventions, layout or indentation is not be covered by the document.

For the ISO 26262 clauses allocated to software architecture, unit design and implementation, the document provides an interpretation of how these clauses apply specifically to C++.

In the current release, the following are known limitations:

1. The rule set for parallel computing is not provided.
2. The rule set for security (as long as it is not common to critical software or safety-related software) is not provided.
3. The rule set for C++ standard libraries is partial (incomplete).
4. All remaining non-analyzed rules from CERT and HIC++ are concurrency/security related.
5. The traceability to C++ Core Guidelines contains some non-analyzed rules (although more rules are covered compared to 2018-03).

The limitations will be addressed in future versions of the AUTOSAR Coding Guidelines.

18 Core Types

Core Types defines common classes and functionality used by multiple Functional Clusters as part of their public interfaces. One of the rationale to define Core Types was to include common complex data types that are often used in the interface definition.

18.1 Error Handling

Overview

Handling errors is a crucial topic for any software development. For safety-critical software, it is even more important, because lives can depend on it. However, current standards for the development of safety-critical software impose significant restrictions on the build toolchain, especially with regard to C++ exceptions. For ASIL applications, using C++ exceptions is usually not possible due to the lack of exceptions support with ASIL-certified C++ compilers.

The Adaptive Platform introduces a concept that enables error handling without C++ exceptions and defines a number of C++ data types to aid in this.

From an application programmer's point of view, the central types implementing this concept are `ara::core::ErrorCode` and `ara::core::Result`.

ErrorCode

An instance of `ara::core::ErrorCode` represents a specific error condition within a software. It is similar to `std::error_code`, but differs in significant aspects from it.
An `ErrorCode` always contains an enumeration value (type-erased into an integral type) and a reference to an *error domain*. The enumeration value describes the specific type of error, and the error domain reference defines the context where that error is applicable. Additional optional members are a user-defined message string and a vendor-defined supplementary error description value.

Within the Adaptive Platform, each Functional Cluster defines one or more error domains. For instance, the Functional Cluster "Core Types" defines the error domain "Posix", which contains error codes that have been defined by POSIX.1. These are equivalent to `std::errc` from the C++11 standard.

Result

Class `ara::core::Result` is a wrapper type that either contains a value or an error. Due to its templated nature, both value and error can be of any type. However, the error type is defaulted to `ara::core::ErrorCode`, and it is expected that this assignment is kept throughout the Adaptive Platform.

Because the error type has a default, most declarations of `ara::core::Result` only need to give the type of the value, e.g. `ara::core::Result<int>` for a `Result` type that contains either an `int` or an `ara::core::ErrorCode`.

The contained value or error can be accessed via the member functions `Result::Value` or `Result::Error`. It is the caller's responsibility to ensure that these access functions are called only if the `Result` instance contains a value or an error, respectively. The type of the content of a `Result`, i.e. a value or an error, can be queried by `Result::HasValue`. None of these member functions throw any exceptions and thus can be used in environments that do not support C++ exceptions.

In addition to the exception-less workflow described above, the class `ara::core::Result` allows to convert a contained `ara::core::ErrorCode` object into a C++ exception, by calling `ara::core::Result::ValueOrThrow`. This call returns any contained value as-is, but treats a contained error by throwing the corresponding exception type, which is automatically derived from the contents of the contained `ara::core::ErrorCode`.

Future and Promise

Similar to the way `ara::core::Result` is used as a generalized return type for synchronous function calls, `ara::core::Future` is used as a generalized return type for asynchronous function calls.

`ara::core::Future` is closely modeled on `std::future`, but has been extended to interoperate with `ara::core::Result`.
Similar to `ara::core::Result`, `ara::core::Future` is a class that either contains a value or an error. This content can be extracted in two ways:

1. by calling `ara::core::Future::get`, which returns the contained value, if it exists, or throws an exception otherwise
2. by calling `ara::core::Future::GetResult`, which returns a `ara::core::Result` object which contains the value or the error from the `Future`

Both of these calls will block until the value or error has been made available by the asynchronous function call.

18.2 Advanced data types

In addition to AUTOSAR-devised data types, which are mentioned in the previous section, the Adaptive Platform also contains a number of generic data types.

Some of these types are already contained in the C++11 standard; however, types with almost identical behavior are re-defined within the `ara::core` namespace. The reason for this is that the memory allocation behavior of the std:: types is often

unsuitable for automotive purposes. Thus, the `ara::core` ones define their own memory allocation behavior.
Examples of such data types are `vector`, `map`, and `string`.

Other types defined in `ara::core` have been defined in or proposed for a newer C++ standard, and the Adaptive Platform includes them into the `ara::core` namespace, because they are necessary for supporting certain constructs of the Manifest, or because they are deemed extremely useful to use in an API.
Examples of such data types are `string_view`, `span`, `optional`, and `variant`.

18.3 Primitive data types

Another document, AUTOSAR_SWS_AdaptivePlatformTypes, exists, which defines primitive types that can be used in ServiceInterface descriptions. This document may be considered to be merged with Core Types document in the future.

19 References

[1] Glossary, AUTOSAR_TR_Glossary.pdf.

[2] Main Requirement, AUTOSAR_RS_Main.pdf.

[3] Methodology for Adaptive Platform, AUTOSAR_TR_AdaptiveMethodology.pdf.

[4] FCDesign IAM, AUTOSAR_EXP_FCDesignIdentityAndAccessManagement.pdf.

[5] Design guidelines for using parallel processing technologies on Adaptive Platform, AUTOSAR_EXP_ParallelProcessingGuidelines.pdf.

[6] P. Kruchten, "Architectural Blueprints—The “4+ 1” View Model of Software Architecture," *IEEE Software,* vol. 12, no. 6, pp. 42-50, November 1995.