Document Title	Explanations 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	17-03

Document Change History

Date	Release	Changed by	Change Description
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 – landscape of intelligent ECUs

Today's 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 fundamentally during vehicle life-time.

Future 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 changed 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 well. 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 on-vehicle network has led to 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 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 element is becoming orders of magnitude faster and efficient than legacy inter-ECU communications. This has been made possible by new type of processor inter-connect technologies such as Network-on-Chip (NoC). Such combined effect 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 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.

2.3.1 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 the faster adaptation of novel algorithms and improve application development productivity, if properly employed.

2.3.2 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 set of services, in which may use another in turn, and applications that uses 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 local ECU that the application 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 the 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.

2.3.3 Parallel processing

The distributed computing is inherently parallel. The SOA, as different applications uses different set of services, shares this characteristic. The advancement or manycore processors and heterogeneous computing that offer parallel processing capability offers technological opportunities to harness the compute 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.

2.3.4 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 superficially is not easy to understand.

2.3.5 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.

2.3.6 Planned dynamics

The AP supports 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 Application 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 for planned dynamics might be:

- Pre-determination of service discovery process
- Restriction of dynamic memory allocation to startup phase only
- Fair scheduling policy instead of 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

2.3.7 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 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.

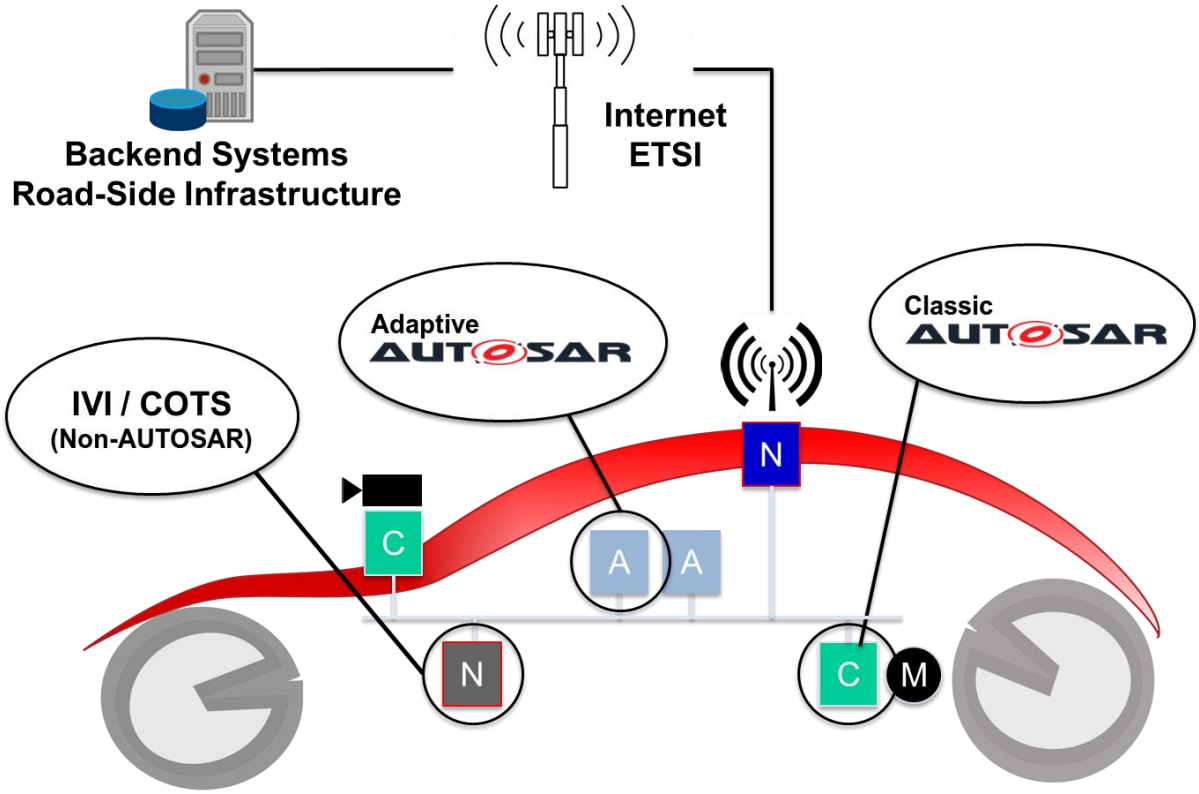

Figure 2-1 Exemplary deployment of different platforms

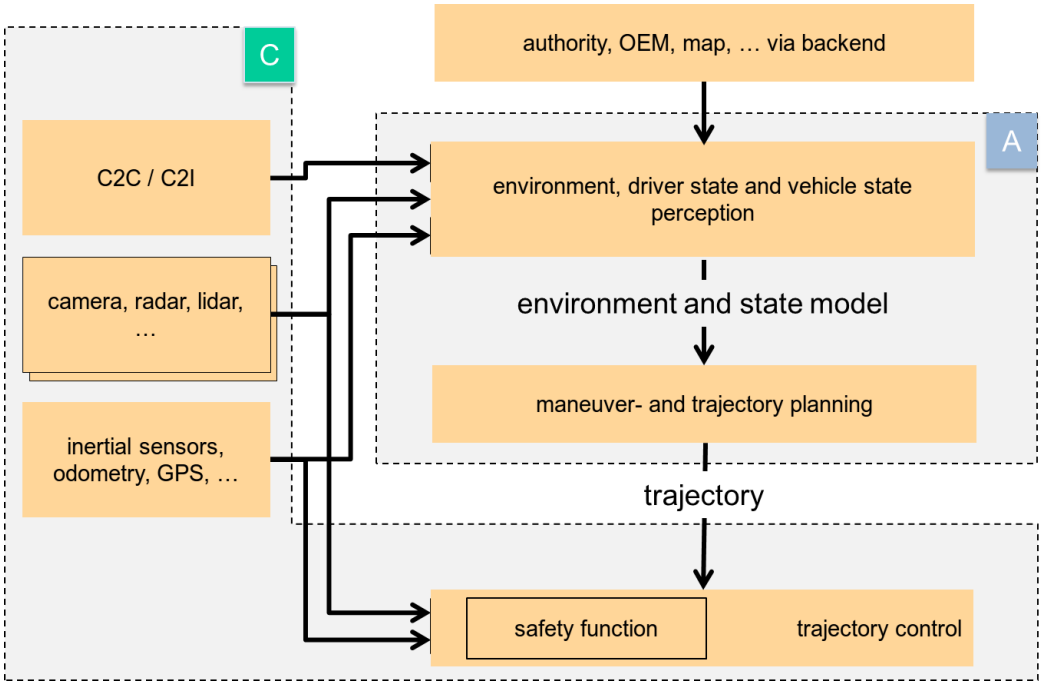

Figure 2-2 Exemplary interactions of AP and CP

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

3.1.1 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 platform standard services called Adaptive Platform Services. 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 and it is up to the design of AP implementation.

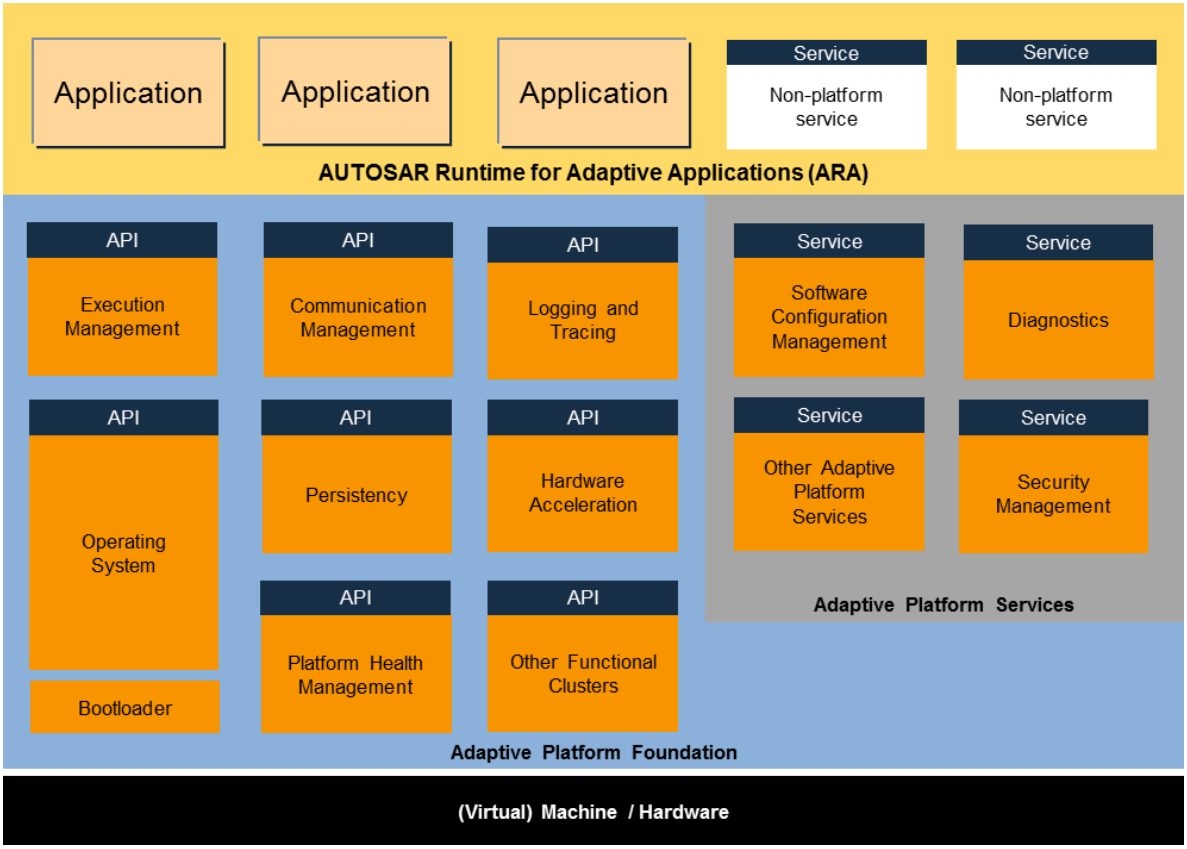

Figure 3-1 AP architecture logical view

Be aware that Figure 3-1 AP architecture logical view contains Functional Clusters that are not part of initial releases 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.

3.1.2 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 thread scheduling policy. In such cases, combined use of both interfaces may be necessary.

3.1.3 Application launch and shutdown

Lifecycles of applications are managed by Execution Management (EM). Loading/launching of 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.

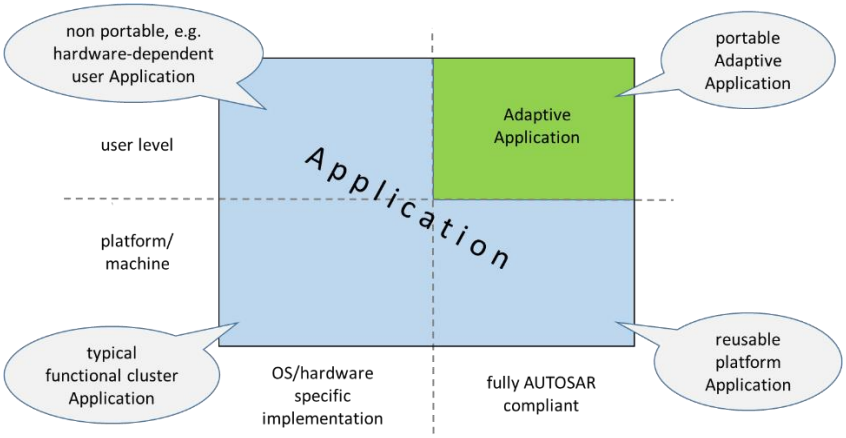

Figure 3-2 Applications

3.1.4 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 explicit communication interface but just a byproduct of functionalities provided by the respective ARA interfaces.

3.1.5 Non-standard interfaces

AA and Functional Clusters may use any non-standard interfaces, if they do not conflict and cope with the standard AP functionalities. 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 of AP is discussed here. Note that the most of contents in this section are for illustration purpose only, and do not constitute the formal requirement specification of AP, as the internals of AP are implementation defined. Any formal requirement on the AP implementation is explicitly stated.

3.2.1 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 name space. 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.

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 are no boundaries 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.

3.2.2 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 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 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.

3.2.3 Interaction between Functional Clusters

In general, the Functional Clusters may interact 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.

3.2.4 Machine/hardware

The AP regards a hardware it runs on as a **Machine**. The rationale behind is that the hardware may be virtualized using various hypervisor related technologies, and to achieve consistent platform view regardless of such.

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.

4 Methodology and Manifest

The support for distributed, independent, and agile development of functional applications requires a standardized approach on 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 4.1 illustrates a draft overview how adaptive methodology might be implemented. For the details of these steps see [3].

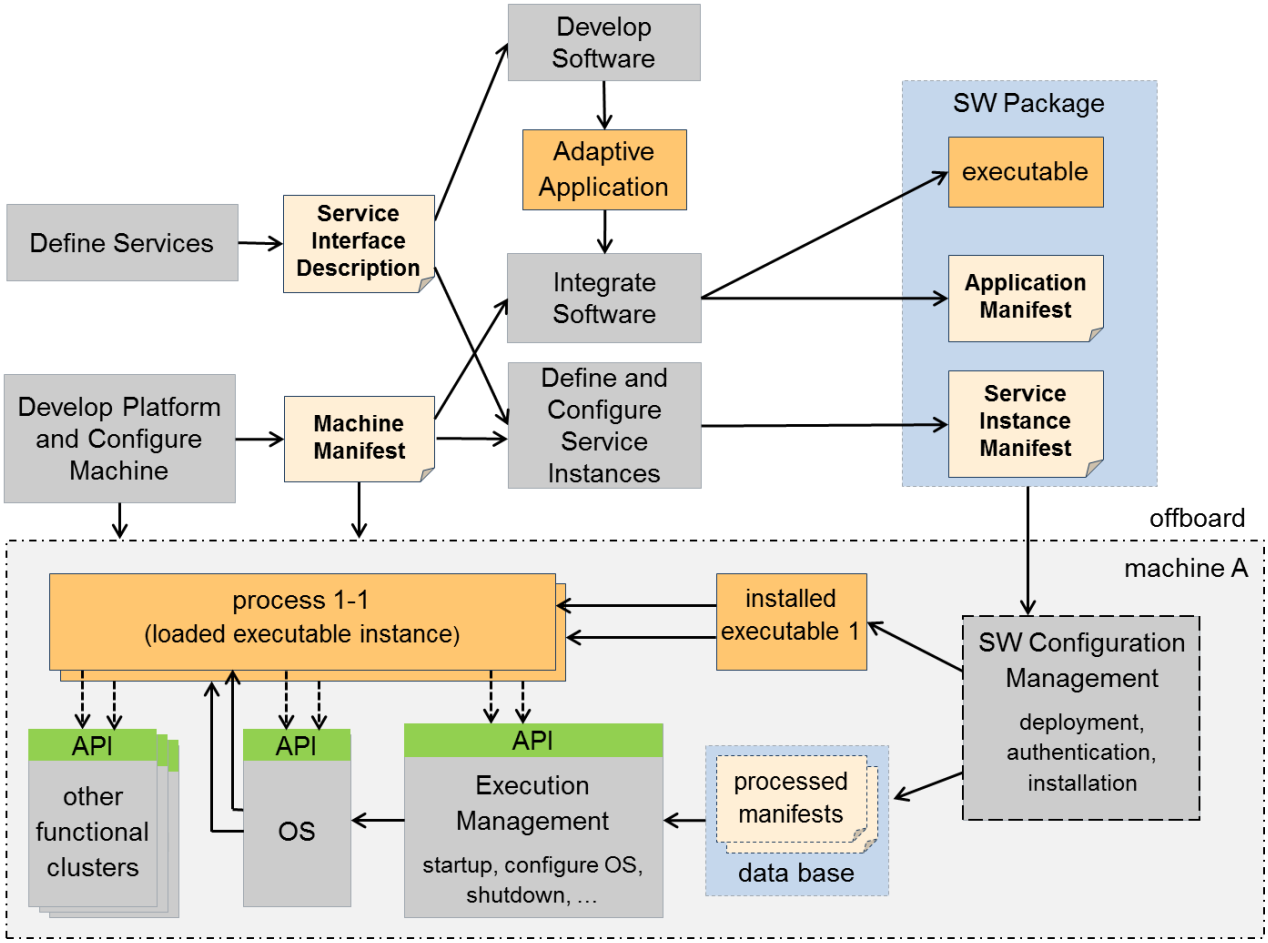

Figure 4-1 AP development workflow

4.1 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 Application Manifest and Service Instance Manifest.

Application Manifest This kind of Manifest is used to specify the deployment-related information of applications running on the AUTOSAR AP.
An Application 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.

4.2 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
- Grouping of applications in order to ease the deployment of software.

The artifacts defined in the application manifest are designed to be independent from a specific deployment of the application software and thus ease the reuse of application implementations for different deployment scenarios.

4.3 Application Manifest

The purpose of the application 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 application 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 Application 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.
- Serialization properties to define the characteristics how data shall be serialized for the transport on the network.
  This is especially interesting for the SOME/IP serialization and the interaction with the AUTOSAR CP (in case serialization properties have been defined on the classic platform they need to be respected on the adaptive platform to support the interaction of the two).

4.4 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 has to 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.

4.5 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
- Configuration of the adaptive platform functional cluster implementations (e.g. the operating system provides a list of OS users with specific rights).
- Documentation of available hardware resources (e.g. how much RAM is available; how many processor cores are available)

5 Operating System

5.1 Overview

The Operating System is responsible for run-time resource management (including time) for all Applications on the Adaptive Platform. Execution Management is responsible for platform initialization and the start-up / shut-down of Applications, working in cooperation with OS.

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 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 application's 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.

5.2 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.

5.3 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 limitation that this may not be portable across different AP implementations.

5.4 Memory management

One of the reasons behind the multi-process support is to realize ‘freedom of interferences’ among different Functional Clusters and AA. As each process has its own address space where the addresses where code and data are located may or may not correspond to their underlying physical storage address is the process’s address space is virtualized. Two instances of the same executable may 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 would be in different physical pages in memory.

5.5 Device management

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

6 Execution Management

6.1 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.

6.2 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 Application Manifest information.

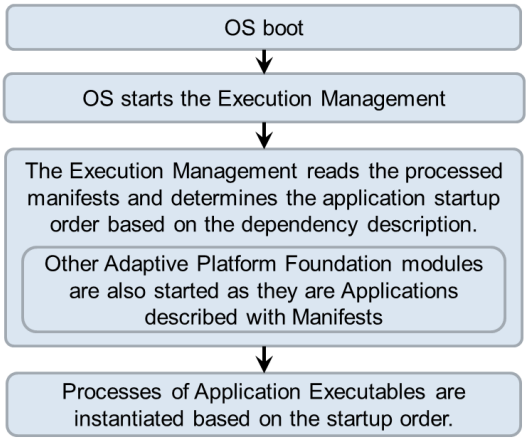

Figure 6-1 AP startup sequence

6.3 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 Application Manifests and derives an ordering for startup/shutdown based on declared Application dependencies. Depending on the Machine State, 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 Application Manifests.

6.4 Machine State Management

Machine State Management provides a mechanism to define the state of the operation for an Adaptive Platform. The Application Manifest allows definition in which Machine State the Application Executables **shall** run. Machine State Management grants full control over the set of Applications to be executed and ensures that Applications are only executed (and hence resources allocated) when needed.

Machine State defines the current set of running Applications. It is significantly influenced by vehicle-wide events and modes. Each Application declares in its Application Manifest in which Machine States it shall be active. There are several mandatory machine states that must be present on each machine. Additional Machine States can be defined on a machine specific basis and are therefore not standardized.

Machine State Management is the ability to control the Machine State during the runtime of an Adaptive AUTOSAR ECU. It is machine specific and AUTOSAR decided against specifying functionality like the Classic Platform’s BswM for the Adaptive Platform. Therefore, Machine State Management can be implemented in two different ways: either integrated directly in the Execution Management or as a separate Machine State Management Application.

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.

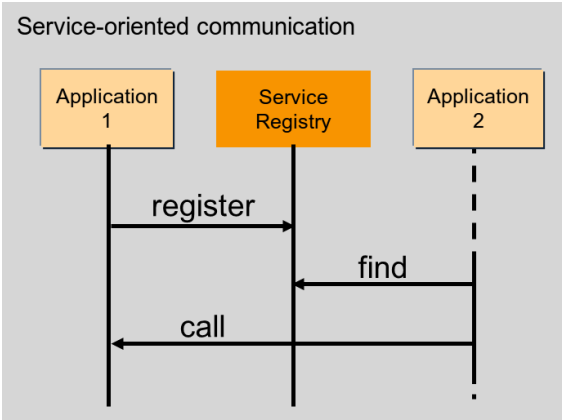

Figure 7-1 Service-oriented communication

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.

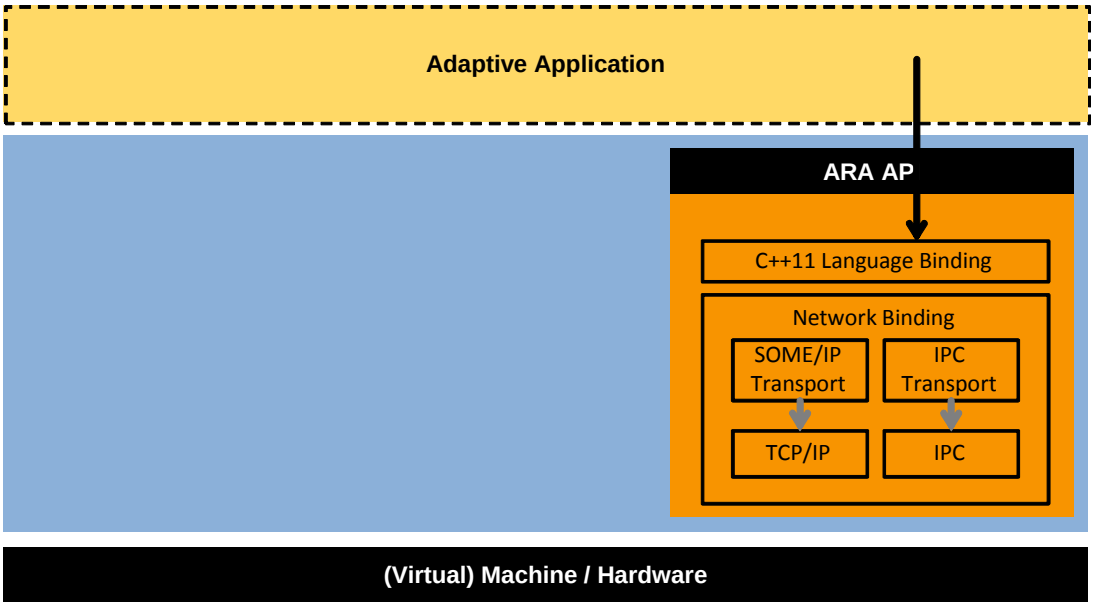

Figure 7-2 Example Language and Network Binding

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 C++ standard template library (std::future).

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.

7.5 Static and dynamic configuration

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.

Note: For Release 17-03 only full service discovery configuration option is available.

8 Diagnostics

8.1 Overview

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

The Diagnostic Management is an Adaptive Platform Service using ARA::COM. Therefore, it is language independent and may be able to serve Adaptive Applications with other language bindings e.g. Java in 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.

8.2 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.

Beside the pseudo parallel client handling of ISO 14229-1, the 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 agnostic Adaptive Application (AA)
In this case the DM uses an existing interface of an AA to fulfill a certain diagnostic request.

The AAs interface, which DM uses to serve the diagnostic request, is an existing one, which cannot be changed and was NOT developed with diagnostic needs in mind. Adapting the existing interface in a way, which perfectly suits the diagnostic requirements/expectations defined by the tester is not an option, since AA shall not be changed.

The typical candidates for such use cases are UDS DataServices (RDBI), where some information provided by an AA anyway like vehicle speed shall be made accessible via UDS diagnostics.

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 SI for a routine control consists of methods "start", "requetsResults" 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 of type UINT8-Array to the service method.

Parameters 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.

8.3 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 enable 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.

9 Persistency

9.1 Overview

Persistency offers mechanisms to Applications to store information in the non-volatile memory of an Adaptive Platform Instance. The data is available over boot and ignition cycles. Persistency offers a library based approach to access the non-volatile memory.

The Persistency library takes storage location identifiers as parameters from the application to address different storage locations.

It is planned to model the names of the used storage locations in the Application Manifest. On the one hand this approach enables an integrator to remap these symbolic storage location identifiers to real locations (e.g. files in a file system or external devices). On the other hand, it is possible to setup storage locations for Persistency during application deployment to fulfill the PSE51 requirements not to create or delete files during runtime. Currently the used storage location identifiers are simply file names.

The available storage locations fall into two categories:

- Key-Value Storage
- Stream Storage

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

9.2 Key-Value Storage

Key-Value Storage provides a mechanism to store and retrieve multiple Key-Value pairs in one storage location. The supported value types are base types, PODs (C++ Plain Old Data structures) and arrays/containers derived from these types.

Adding AUTOSAR data types which are defined in the AUTOSAR model is planned.

9.3 Stream Storage

To support raw access to file like structures Persistency also offers stream mechanisms which are used like the well-known fstream classes in the C++ Standard Library.

10 Safety

10.1 Overview

Safety offers mechanisms to Adaptive Applications to protect the exchange of information inside the vehicle and with the external world. This will include mechanisms for inter- and intra-ECU communication. For this purpose, mechanisms provided will allow fault detection if any corruption has occurred. No mechanisms will be provided to guarantee the integrity of data.

Safety also offers mechanisms to monitor the correct execution of platform functionalities and Adaptive Applications. This allows a defined handling of detected deviations. For this purpose, mechanisms provided will allow fault detection. No mechanisms will be provided to guarantee the integrity of applications.

In addition, Safety offers guidelines such as coding guidelines, which facilitate the safe and secure usage of complex languages like C++.

For the other Functional Clusters (e.g. Execution Management) guidance on possible safety implications will be provided and necessary enhancements will be integrated into the corresponding Functional Clusters.

In general, Safety will provide concepts and documents that will support the development of an Adaptive Platform as Safety Element out of context. For example, there will be an overview of the provided safety features that will be supported by the Adaptive Platform. Safety expectations of system integrators will be addressed and so the development of safety cases will be supported.

In a first step, the focus will be on safety mechanisms for fail-safe systems, but it will be extended to mechanisms for fail-operational systems in future.

10.2 Protection of information exchange (E2E-Protection)

Latest E2E profiles within AUTOSAR will be supported to allow safe communication with Classic Platform ECUs. Where useful, mechanisms will be provided to allow safe communication using more capabilities of the service oriented approached within the Adaptive Platform.

10.3 Watchdog functionality

In a first step, mechanisms will be provided to support fail-safe applications. The following aspects will be considered:

- Alive supervision
- Deadline supervision
- Logical supervision
- Error handling of supervision errors

10.4 C++ coding guidelines

Based on the elaboration of common coding guidelines, like MISRA, HIC, CERT, the C++ Core Guideline, and additional investigations a guideline for the usage of C++14 in safety-related SW-development will be provided.

11 References

[1] Glossary, AUTOSAR_TR_Glossary.pdf.
[2] Main Requirement, AUTOSAR_RS_Main.pdf.
[3] Methodology for Adaptive Platform, AUTOSAR_TR_AdaptiveMethodology.pdf.