Document Title	Specification of Execution Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	721

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	17-03

Document Change History			
Date	**Release**	**Changed by**	**Description**
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction and functional overview

This document is the software specification of the Execution Management functional cluster within the Adaptive Platform.

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. This document describes how these concepts are realized within the Adaptive Platform. Furthermore, the Application Programming Interface (*API*) of the Execution Management is specified.

1.1 What is Execution Management?

Execution Management is responsible for the startup and shutdown of Applications based on Manifest information. The usage of Execution Management is limited to the Adaptive Platform however the latter is usually not exclusively used within a single AUTOSAR System. The vehicle is also equipped with a number of ECUs developed on the *AUTOSAR Classic Platform* and the system design for the entire vehicle will therefore have to cover both ECUs built using that as well as the Adaptive Platform.

1.2 Interaction with AUTOSAR Runtime for Adaptive

The Execution Management is a functional cluster contained in the Adaptive Platform Foundation. The set of programming interfaces to the Adaptive Applications is called ARA.

Execution Management, in common with other Adaptive Applications is assumed to be a process executed on a POSIX compliant operating system. Execution Management is responsible for initiating execution of the processes in all the Functional Clusters, Adaptive AUTOSAR Services, and Adaptive Applications. The launching order must be given to the Execution Management according to the specification defined in this document to ensure proper startup of the system.

The Adaptive AUTOSAR Services are provided via the Communication Management functional cluster of the Adaptive Platform Foundation. In order to use the Adaptive AUTOSAR Services, the functional clusters in the Foundation must be properly initialized beforehand. Refer to the respective specifications regarding more information on the Communication Management.

2 Acronyms and abbreviations

All technical terms used throughout this document – except the ones listed here – can be found in the official [1, AUTOSAR glossary] or [2, TPS Manifest Specification].

Term	Description
Process	A process is a loaded instance of an `Executable` to be executed on a `Machine`.
Application Dependency	–
Execution Management	The element of the `Adaptive Platform` responsible for the ordered startup and shutdown of the `Adaptive Platform` and the `Application`s.
Machine State Management	The element of the `Execution Management` managing modes of operation for `Adaptive Platform`. It allows flexible definition of functions which are active on the platform at any given time.
Machine State	The element of the `Machine State Management` which characterize the current status of the machine. It defines the set of active `Application`s for any certain situation. The set of `Machine State`s is platform specific and it will be deployed in the `Machine Manifest`.

Table 2.1: Technical Terms

3 Related documentation

3.1 Input documents

The main documents that serve as input for the specification of the `Execution Management` are:

[1] Glossary
AUTOSAR_TR_Glossary

[2] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[3] Requirements on Execution Management
AUTOSAR_RS_ExecutionManagement

[4] Requirements on Operating System Interface
AUTOSAR_RS_OperatingSystemInterface

[5] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology

3.2 Related standards and norms

See chapter 3.1.

3.3 Related specification

See chapter 3.1.

4 Constraints and assumptions

4.1 Known limitations

This chapter lists known limitations of `Execution Management` and their relation to this release of the `Adaptive Platform`. The intent is to not only provide a specification of the current state of `Execution Management` but also an indication how the `Adaptive Platform` will evolve future releases.

The following functionality is mentioned within this document but is not fully specified in this release:

- Section 7.3 **Platform Lifecycle Management**, in particular the issues of *Platform monitoring* and *Application cleanup*.
- Figures 7.5 and 7.6 cover Watchdog Management and the `ExecutionManagement::ApplicationRestart` interface.
- Appendix A details requirements from `Execution Management` Requirement Specification that are not elaborated within this specification. The presence of these requirements in this document ensures that the requirement tracing is complete and also provides an indication of how `Execution Management` will evolve in future releases of the `Adaptive Platform`.

The functionality described above is subject to modification and will be considered for inclusion in a future release of this document.

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other modules

5.1 Platform dependencies

Operating System Interface

The `Execution Management` functional cluster is dependent on the Operating System Interface [4]. The OSI is used by `Execution Management` to control specific aspects of `Application` execution. E.g. to set scheduling parameters or to execute an `Application`.

5.2 Other dependencies

Currently, no other library dependencies existing.

6 Requirements tracing

The following tables reference the requirements specified in [3] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_EM_00001]	The `Execution Management` shall load `Executable` s.	[SWS_EM_01017]
[RS_EM_00002]	The `Execution Management` shall set-up one process for the execution of each `Executable` instance	[SWS_EM_01014] [SWS_EM_01015] [SWS_EM_01039] [SWS_EM_01040] [SWS_EM_01041] [SWS_EM_01042] [SWS_EM_01043]
[RS_EM_00003]	The `Execution Management` shall support the checking of the integrity of `Executable` s at startup of `Executable` .	[SWS_EM_99999]
[RS_EM_00004]	The `Execution Management` shall support the authentication and authorization of `Executable` s at startup of `Executable`	[SWS_EM_99999]
[RS_EM_00005]	The `Execution Management` shall support the configuration of OS resource budgets for `Executable` and groups of `Executable` s	[SWS_EM_99999]
[RS_EM_00006]	The `Execution Management` shall support the analysis of available and required OS resource budgets for `Executable` s and groups of `Executable` s during installation and run-time	[SWS_EM_99999]
[RS_EM_00007]	The `Execution Management` shall support of the allocation of dedicated resources for the `Executable` (e.g GPU)	[SWS_EM_99999]
[RS_EM_00008]	The `Execution Management` shall support the binding of `Executable` threads to a specified set of processor cores.	[SWS_EM_99999]
[RS_EM_00009]	Only `Execution Management` shall start `Executables`	[SWS_EM_01030]
[RS_EM_00010]	The `Execution Management` shall support multiple instantiation of `Executable` s	[SWS_EM_00017] [SWS_EM_01012] [SWS_EM_01033]
[RS_EM_00011]	`Execution Management` shall support self-initiated graceful shutdown of `Executable` instances	[SWS_EM_01005]
[RS_EM_00013]	`Execution Management` shall support configurable recovery actions	[SWS_EM_99999]

Requirement	Description	Satisfied by
[RS_EM_00050]	The `Execution Management` shall do a system-wide coordination of activities	[SWS_EM_99999]
[RS_EM_00051]	The `Execution Management` shall provide functions to the `Executable` for configuring external trigger conditions for its activities	[SWS_EM_99999]
[RS_EM_00052]	The `Execution Management` shall provide functions to the `Executable` for configuring cyclic triggering of its activities	[SWS_EM_99999]
[RS_EM_00100]	The `Execution Management` shall support the ordered startup and shutdown of `Executable` s	[SWS_EM_01000] [SWS_EM_01050] [SWS_EM_01051]
[RS_EM_00101]	The `Execution Management` shall provide `Machine State` Management functionality	[SWS_EM_01013] [SWS_EM_01023] [SWS_EM_01024] [SWS_EM_01025] [SWS_EM_01026] [SWS_EM_01027] [SWS_EM_01028] [SWS_EM_01029] [SWS_EM_01034] [SWS_EM_01035] [SWS_EM_01036] [SWS_EM_01037] [SWS_EM_01056] [SWS_EM_01057] [SWS_EM_01058] [SWS_EM_01059] [SWS_EM_01060] [SWS_EM_02005] [SWS_EM_02006] [SWS_EM_02007] [SWS_EM_02008] [SWS_EM_02009] [SWS_EM_02014] [SWS_EM_02019] [SWS_EM_02031]
[RS_EM_00103]	`Execution Management` shall support application lifecycle management	[SWS_EM_01002] [SWS_EM_01003] [SWS_EM_01004] [SWS_EM_01005] [SWS_EM_01006] [SWS_EM_01052] [SWS_EM_01053] [SWS_EM_01055] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02030] [SWS_EM_02031]

7 Functional specification

7.1 Technical Overview

This chapter presents a short summary of the relationship between Application, Executable, and Process, including the configuration information which is assigned to these items. For a detailed and formal specification see [5].

7.1.1 Application

Applications are developed to resolve a set of coherent functional requirements. An Application consists of executable software units, additional execution related items (e.g. data or parameter files), and descriptive information used for integration end execution (e.g. a formal model description based on the AUTOSAR meta model, test cases).

Applications can be located on user level above the middleware or can implement functional clusters of the Adaptive Platform (located on the level of the middleware), see [TPS_MANI_01009] in [2].

Applications might use all mechanisms and APIs provided by the operating system and other functional clusters of the Adaptive Platform, which in general restricts portability to other Adaptive Platforms.

All Applications, including Adaptive Applications (see below), are treated the same by Execution Management.

7.1.2 Adaptive Application

An Adaptive Application is a specific type of Application. The implementation of an Adaptive Application fully complies with the AUTOSAR specification, i.e. it is restricted to use APIs standardized by AUTOSAR and needs to follow specific coding guidelines to allow reallocation between different Adaptive Platforms.

Adaptive Applications are always located above the middleware. To allow portability and reuse, user level Applications should be Adaptive Applications whenever technically possible.

Figure 7.1 shows the different types of Applications.

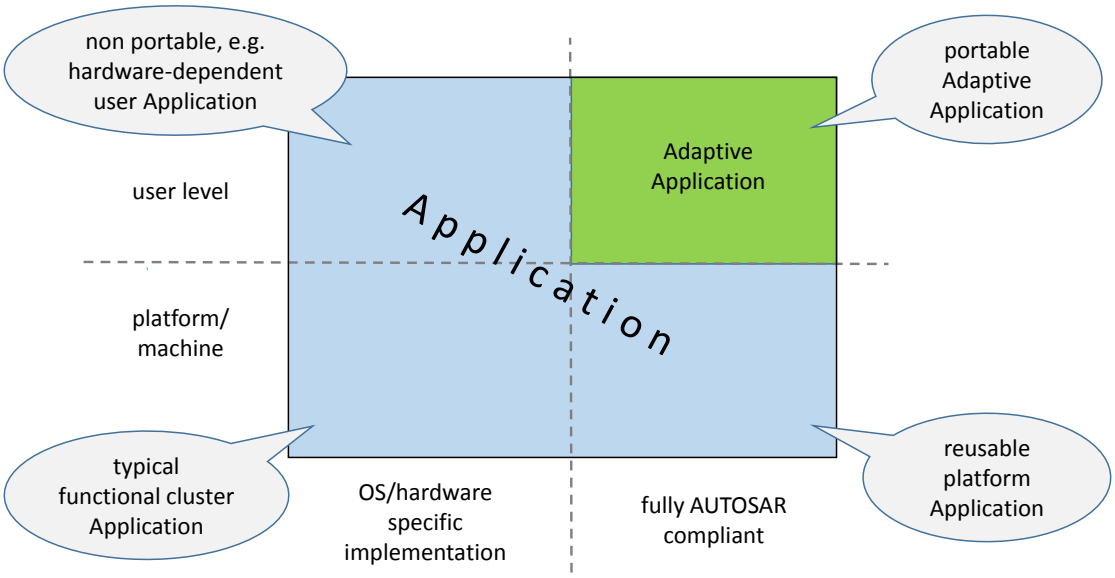

Figure 7.1: Types of Applications

An `Adaptive Application` is the result of functional development and is the unit of delivery for `Machine` specific configuration and integration. The development of an `Adaptive Application` also requires consideration of non-functional (e.g. concerning safety, security, and performance) requirements and constraints. Some contracts (e.g. concerning used libraries) and Service Interfaces to interact with other `Adaptive Application`s need to be agreed on beforehand. For details see [5].

An `Adaptive Application` might be configurable, i.e. an output of development can contain a range for configuration settings, which need to be fixed at integration or calibration time to meet the specific system requirements. As an example, a video filter can be developed to support different video resolutions. At integration time, the specific resolution is known and memory resources can be configured.

To allow tool-supported integration (without extensive human intervention) of an `Adaptive Application`, the Service Interfaces, the component model, properties (e.g. concerning safety or performance), fixed configuration settings and variable configuration ranges should be delivered with the `Adaptive Application` in a machine readable form, preferably based on the AUTOSAR meta model. This also allows checking whether an additional `Adaptive Application` can be integrated for a specific `Machine`. In general, because one should assume a potentially high number of differently configured `Machine`s, e.g. due to an individual selection of `Adaptive Application`s by the end-user, support for a fully automatic integration process might be necessary.

7.1.3 Executable

An Executable is a software unit which is part of an Application. It has exactly one entry point (main function), see [SWS_OSI_01300]. An Application can be implemented in one or more Executables.

The lifecycle of Executables usually consists of:

Process Step	Code Output	Model Description Output
Development	Portable, target-independent (assuming agreed on contracts and Service Interfaces are considered) source code	Standardized description of Application design properties and demands (component model, Service Interfaces, trigger conditions, timing and monitoring assumptions, resources, safety assumptions etc.), range for configuration options.
Delivery	Compiled code for delivery into the integration process step, in case the source code shall not be revealed to the integrating organization. Additional target-specific information (e.g. type of microprocessor) is required.	Same as above plus additional build chain information.
Integration	Linked, configured and calibrated binary for deployment onto the target Machine. The binary might contain code which was generated at integration time.	Application Manifest, see 7.1.5 and [2].
Installation	Binary installed on the target Machine.	Processed Application Manifest, stored in a platform-specific format which is efficiently readable at Machine startup.
Execution	Process started as instance of the binary.	The Execution Management uses contents of the Processed Application Manifest to start up and configure each process individually depending on the current Machine State (see chapter 7.6).

Table 7.1: Executable Lifecycle

Executables which belong to the same Adaptive Application might need to be deployed to different Machines, e.g. to one high performance Machine and one high safety Machine.

Remark: Throughout this document, on execution level the term Application refers to one Executable of this Application, i.e. whenever mechanism on the Machine or contents of the Application Manifest are described, there is no distinction between Application and Executable, because the Application component model is flattened into independent Executables after deployment.

7.1.4 Process

[SWS_EM_00017] Application Processes ⌈ Each `Executable` instance, i.e. each process, has its own specific set of startup parameters defined by [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01014], [TPS_MANI_01015], [TPS_MANI_01059], [TPS_MANI_01017] and [TPS_MANI_01041]. ⌋*(RS_EM_00010)*

Remark: In this release of this document it is assumed, that processes are self-contained, i.e. that they take care of controlling thread creation and scheduling by calling APIs from within the code. `Execution Management` only starts and terminates the processes and while the processes are running, `Execution Management` only interacts with the processes by using `State Management` mechanisms (see 7.6).

7.1.5 Application Manifest

The `Application Manifest` consists of parts of the `Application` design information which is provided by the application developer in the application description, and additional machine-specific information which is added at integration time. For details on the `Application Manifest` contents see chapter 7.5. A formal specification can be found in [2] .

An `Application Manifest` is created at integration time and deployed onto a `Machine` together with the `Executable`s it is attached to. It describes in a standardized way the machine-specific configuration of Process properties (startup parameters, resource group assignment, Service Interfaces, priorities etc.).

Each instance of an `Executable` binary, i.e. each started process, is individually configurable, with the option to use a different configuration set per `Machine State` (see 7.6 and [TPS_MANI_01012]).

For deployment, the configuration information of several instances of the same or different `Executable` binaries can be combined in the same Manifest file, if the instances are started on the same `Machine`.

7.1.6 Machine Manifest

The `Machine Manifest` is also created at integration time for a specific `Machine` and is deployed like `Application Manifest`s whenever its contents change. The `Machine Manifest` holds all configuration information which cannot be assigned to a specific `Executable`, i.e. which is not already covered by an `Application Manifest`.

The contents of a `Machine Manifest` includes the configuration of `Machine` properties and features (resources, safety, security, services etc.), e.g. available states, resource groups, access right groups, scheduler configuration, SOME/IP configuration, memory segmentation.

7.1.7 Manifest format

The Application Manifests and the Machine Manifest can be transformed into a platform-specific format (called Processed Manifest), which is efficiently readable at Machine startup. The format transformation can be done either off board at integration time or at deployment time, or on the Machine (by SW Configuration Management) at installation time.

7.2 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 started as part of the Adaptive Platform startup phase and is responsible for the initialization of the Adaptive Platform and deployed Applications.

   During execution, Execution Management monitors the Adaptive Platform and, when required, the ordered shutdown of the Adaptive Platform.

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. The dependencies are described in the Application Manifests, see [TPS_MANI_01041].

   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.

7.3 Platform Lifecycle Management

The Execution Management controls the ordered startup and shutdown of the Adaptive Platform. The Platform Lifecycle Management characterize different stages of the Adaptive Platform including:

Platform startup – the Execution Management is started as the "init" process by the Operating System and then takes over responsibility for subsequent initialization of the Adaptive Platform and deployed Application Executables.

[SWS_EM_01030] Start of Application execution ⌈ Execution Management shall be solely responsible for initiating execution of Applications. ⌋ *(RS_EM_00009)*

Note that [SWS_EM_01030] is exclusive; once the Execution Management is running no other element of Adaptive Platform initiates Application execution.

Platform monitoring – the Execution Management works in conjunction with the Watchdog to perform Application monitoring and to "clean-up" should an Application terminate unexpectedly. Note: This functionality is not fully specified, see 4.1.

Platform shutdown – the Execution Management performs the ordered shutdown of the Adaptive Platform based on the dependencies, with the exception that already terminated Applications do not represent an error in the order.

7.4 Application Lifecycle Management

7.4.1 Process States

From the execution stand point, *Process State* characterize the lifecycle of any Application Executable. Note that each instance (i.e. process) of an Application Executable is independent and therefore has its own *Process State*.

[SWS_EM_01002] Idle Process State ⌈ The **Idle** Process State shall be the Process state prior to creation of the Application's process and resource allocation. ⌋ *(RS_EM_00103)*

[SWS_EM_01003] Starting Process State ⌈ The **Starting** Process State shall apply when the Application's process has been created and resources have been allocated. ⌋*(RS_EM_00103)*

[SWS_EM_01004] Running Process State ⌈ The **Running** Process State shall apply to an Application's process after it has been scheduled and it has reported Running State to the Execution Management. ⌋*(RS_EM_00103)*

[SWS_EM_01005] Terminating Process State ⌈ The **Terminating** Process State shall apply either after an `Application`'s process has received the termination indication or after it has decided to terminate. ⌋*(RS_EM_00103, RS_EM_00011)*

The termination indication uses the `ReportApplicationState` API (see Section 8.2.1.3).

On entering the **Terminating** Process State the `Application`'s process performs persistent storage of the working data, frees all `Application`'s process internal resources, and exits.

[SWS_EM_01006] Terminated Process State ⌈ The **Terminated** Process State shall apply after the `Application`'s process has been terminated and the process resources have been freed. For that, `Execution Management` shall observe the exit status of all `Application`s processes, with the POSIX `waitpid()` command. From the resource allocation stand point, **Terminated** state is similar to the **Idle** state as there is no process running and no resources are allocated anymore. From the execution stand point, **Terminated** state is different from the **Idle** state since it tells that the `Application`'s process has already been executed and terminated. This is relevant for one shot `Applications` processes which are supposed to run only once. Once they have reached their **Terminated** state, they shall stay in that state and never go back in any other state. E.g. System Initialization `Application`'s process is supposed to run only once before any other application execution. ⌋*(RS_EM_00103)*

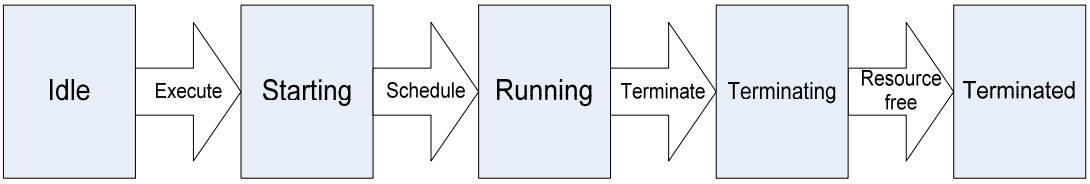

Figure 7.2: Process Lifecycle

7.4.2 Startup and shutdown

[SWS_EM_01050] Start dependent `Application Executables` ⌈ The `Execution Management` shall respect `Application Dependency`s and start any `Application Executable`s in this list first. In case no dependency is specified between two `Application Executable`s, they should be started in parallel. ⌋*(RS_EM_00100)*

[SWS_EM_01051] Shutdown `Application Executables` ⌈ The `Execution Management` shall respect `Application Dependency`s and shutdown dependent `Application Executable`s before the `Application Executable` that was initially requested to be shutdown. ⌋*(RS_EM_00100)*

[SWS_EM_01012] Application Argument Passing ⌈ The `Execution Management` shall provide argument passing for a `Process` containing one ore more `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig`.

At the initiation of startup of a Process, the aggregated StartupOptions of the StartupConfig referenced by the ModeDependentStartupConfig shall be passed to the call of the exec-family based POSIX interface to start the Process by the Operating System, with the following behavior:

- for arg_0, the name of the Application Executable shall be passed
- for each aggregated StartupOption, starting with $n = 1$:
  - for a StartupOption with StartupOption.optionKind = commandLineSimpleForm: arg_n = StartupOption.optionArgument
  - for a StartupOption with StartupOption.optionKind = commandLineShortForm:
    - When multiplicity of StartupOption.optionArgument = 1:

      arg_n = '-' + StartupOption.optionName + ' ' + StartupOption.optionArgument
    - otherwise:

      arg_n = '-' + StartupOption.optionName
  - for a StartupOption with StartupOption.optionKind = commandLineLongForm:
    - When multiplicity of StartupOption.optionArgument = 1:

      arg_n = '--' + StartupOption.optionName + '=' + StartupOption.optionArgument
    - otherwise:

      arg_n = '--' + StartupOption.optionName
  - $n = n + 1$

⌋(*RS_EM_00010*)

7.4.3 Startup sequence

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[1]. 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 to launch Adaptive Applications.

[SWS_EM_01000] Startup order ⌈ The startup order of the platform-level Applications and also the Adaptive Applications are determined by the Execution

[1] Typically the *init* process

Management, based on Machine Manifest and Application Manifest information. ⌋(*RS_EM_00100*) Please see Section 7.5.1.

Figure 7.3 shows the overall startup sequence.

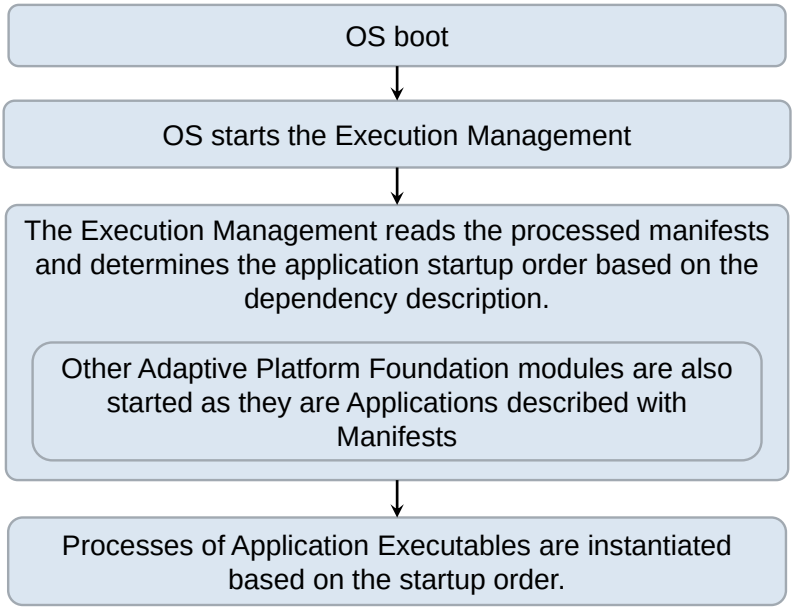

Figure 7.3: Startup sequence

7.4.3.1 Application dependency

The Execution Management provides support to the Adaptive Platform for ordered startup and shutdown of Applications. This ensures that Applications are started before dependent Applications use the services that they provide and, likewise, that Applications are shutdown only when their provided services are no longer required. At development or integration time, the service dependencies need to be mapped to dependencies between the Executable instances which provide or require these services. These dependencies, see [TPS_MANI_01041], are configured in the Application Manifests, which is created at integration time based on information provided by the Application developer.

In real life, specifying a simple dependency to an Application might not be sufficient to ensure that the depending service is actually provided. Since some Applications shall reach a certain *Application State* to be able to offer their services to other Applications, the dependency information shall also refer to *Application State* of the Application specified as dependency. With that in mind, the dependency information may be represented as a pair like: <Application>.<ApplicationState>. For more details regarding the *Application States* refer to Section 7.6.2.

The following dependency use-cases have been identified:

- In case Application B has a simple dependency on Application A, the *Running Application State* of Application A is specified in the dependency section of Application's B Application Manifest.
- In case Application B depends on One-Shot Application A, the *Terminated Application State* of Application A is specified in the dependency section of Application's B Application Manifest.

Version information within the Application Manifest is required since a consuming Executable and its required services might not be compatible with all versions of the producing Executable and its provided services. An example for the definition of the version information attached to several Executables could be found in Listing 7.1.

Listing 7.1: Example for Executable versions

```
<AR-PACKAGE>
  <SHORT-NAME>Executables</SHORT-NAME>
  <ELEMENTS>
    <EXECUTABLE>
      <SHORT-NAME>RadarSensorVR</SHORT-NAME>
      <VERSION>1.0.3</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>RadarSensorVL</SHORT-NAME>
      <VERSION>1.0.4</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>Diag</SHORT-NAME>
      <VERSION>1.0.0</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>SensorFusion</SHORT-NAME>
      <VERSION>1.0.2</VERSION>
    </EXECUTABLE>
  </ELEMENTS>
</AR-PACKAGE>
```

An example for the definition of the Executable dependency information could be found in Listing 7.2

Listing 7.2: Example for Executable dependency

```
<PROCESS>
  <SHORT-NAME>SensorFusion</SHORT-NAME>
  <EXECUTABLE-REF DEST="EXECUTABLE">/Executables/SensorFusion</EXECUTABLE-
      REF>
  <MODE-DEPENDENT-STARTUP-CONFIGS>
    <MODE-DEPENDENT-STARTUP-CONFIG>
      <EXECUTION-DEPENDENCYS>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
                DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVR/
                ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
                PROTOTYPE-REF>
```

```
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
               DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVL/
               ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
               PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
               DECLARATION-GROUP-PROTOTYPE">/Processes/Diag/
               ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
               PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
      </EXECUTION-DEPENDENCYS>
      <STARTUP-CONFIG-REF DEST="STARTUP-CONFIG">/StartupConfigSets/
         StartupConfigSet_AA/SensorFusion_Startup</STARTUP-CONFIG-REF>
    </MODE-DEPENDENT-STARTUP-CONFIG>
  </MODE-DEPENDENT-STARTUP-CONFIGS>
</PROCESS>
```

7.5 Handling of Application Manifest

7.5.1 Overview

The Application Manifest is created at integration time by the system integrator. It contains information provided by the Application developer, which has been adapted to the Machine-specific environment, and additional attributes and other model elements.

An Application Manifest includes all information needed for deployment and installation of Application Executables onto an Adaptive Platform and execution of its instances (i.e. processes). The Execution Management is responsible for parsing the content of the Application Manifests to perform integrity checks over the available data, to determine Machine State and startup dependencies, and to configure the Operating System accordingly at startup of the Executable instances.

For more information regarding the Application Manifest specification please see [2].

To perform its necessary actions, the Execution Management imposes a number of requirements on the content of the Application Manifest. This section serves as a reference for those requirements.

7.5.2 Application Dependency

The required dependency information is provided by the Application developer. It is adapted to the specific Machine environment at integration time and made available in the Application Manifest.

The Execution Management parses the information and uses it to build the startup sequence to ensure that the required antecedent Executable instances have reached a certain *Application State* before starting a dependent Executable instance.

7.5.3 Application Arguments

The set of static arguments required by an Application Executable can either be provided by the Application developer or specified at integration time. The integrator then makes the arguments available in the Application Manifest for use by Execution Management when starting the Application Executable's process.

7.5.4 Machine State

[SWS_EM_01013] Machine State ⌈ The Execution Management shall support the execution of specific instances of Application Executables depending on the current Machine State, based on information provided in the Application Manifests. ⌋*(RS_EM_00101)*

Each instance of an Application Executable is assigned to one or several startup configurations (StartupConfig), which each can define the startup behaviour in one or several Machine States. For details see [2]. By parsing this information from the Application Manifests, Execution Management can determine which processes need to be launched if a specific Machine State is entered, and which startup parameters are valid.

[SWS_EM_01033] Application start-up configuration ⌈ To enable an Application Executable to be launched in multiple Machine States, Execution Management shall be able to configure the Application start-up on every Machine State change based on information provided in the Application Manifest. ⌋*(RS_EM_00010)*

7.5.5 Scheduling Policy

[SWS_EM_01014] Scheduling policy ⌈ Execution Management shall support the configuration of the scheduling policy when lauching an instance of an Application Executable, based on information provided by the Application Manifest. ⌋(*RS_EM_00002*)

For the detailed definitions of these policies, refer to The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. Note, SCHED_OTHER shall be treated as non-realtime scheduling policy, and actual behavior of the policy is implementation specific. It must not be assumed that the scheduling behavior is compatible between different Adaptive Platform implementations, except that it is a non-realtime scheduling policy in a given implementation.

- **[SWS_EM_01041] Scheduling FIFO** ⌈ The Execution Management shall be able to configure FIFO scheduling using policy SCHED_FIFO. ⌋(*RS_EM_00002*)
- **[SWS_EM_01042] Scheduling Round-Robin** ⌈ The Execution Management shall be able to configure round-robin scheduling using policy SCHED_RR. ⌋ (*RS_EM_00002*)
- **[SWS_EM_01043] Scheduling Other** ⌈ The Execution Management shall be able to configure non real-time scheduling using policy SCHED_OTHER. ⌋ (*RS_EM_00002*)

7.5.6 Scheduling Priority

[SWS_EM_01015] Scheduling priority ⌈ Execution Management shall support the configuration of a scheduling priority when lauching an instance of an Application Executable, based on information provided by the Application Manifest. ⌋ (*RS_EM_00002*)

The available priority range and actual meaning of the scheduling priority depends on the selected scheduling policy.

[SWS_EM_01039] Scheduling priority range for SCHED_FIFO and SCHED_RR ⌈ For SCHED_FIFO ([SWS_EM_01041]) and SCHED_RR ([SWS_EM_01042]), an integer between 1 (lowest priority) and 32 (highest priority) shall be used. ⌋(*RS_EM_00002*)

[SWS_EM_01040] Scheduling priority range for SCHED_OTHER ⌈ For the non real-time policy SCHED_OTHER ([SWS_EM_01043]) the scheduling priority shall always be zero. ⌋(*RS_EM_00002*)

7.5.7 Application Binary Name

[SWS_EM_01017] Application Binary Name ⌈ The `Execution Management` shall obtain the name of the `Application Executable` from the `Application Manifest`. ⌋*(RS_EM_00001)*

7.6 State Management

7.6.1 Overview

`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 Executable`s have to run (see [2]). `Machine State Management` grants full control over the set of `Application`s to be executed and ensures that `Application`s are only executed (and hence resources allocated) when actually needed.

Two different states are relevant for `Execution Management`:

- Application State
- Machine State

They are introduced within this chapter.

7.6.2 Application State

The *Application State* characterizes the lifecycle of any instance of an `Application Executable`. The states are defined by the `ApplicationState` enumeration (see Section 8.1.1).

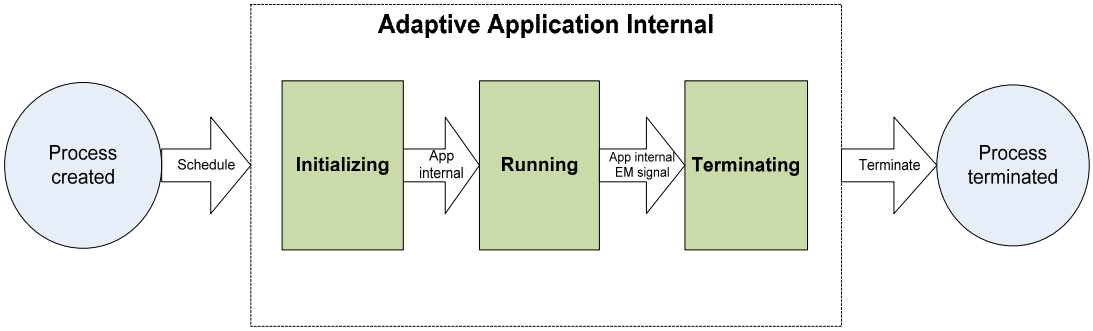

Figure 7.4: Adaptive Application States

[SWS_EM_01052] Application State `Initializing` ⌈ When launched by `Execution Management`, an `Application` shall set its state to `Initializing` (represented by the `kInitializing` enumeration) prior to commencing its initialization procedure where `Application` data initialization happens. ⌋*(RS_EM_00103)*

[SWS_EM_01053] Application State `Running` ⌈ Once the initialization is complete, the `Application` shall switch to the `Running` state (represented by the `kRunning` enumeration value). ⌋*(RS_EM_00103)*

Note: An `Application` is expected to perform its main functionality within the code section of the `Running` state.

[SWS_EM_01055] Application State `Termination` ⌈

- The switch from the `Running` state to `Terminating` shall be initiated by the POSIX Signal `SIGTERM` or by any `Application` internal functionality causing this state change.
- On Reception of that Signal, the `Application` shall switch to the `Terminating` state, represented by the `kTerminating` enumeration value.
- During the `Terminating` state, `Application` shall free internally used resources.
- When the `Terminating` state finishes, the `Application` shall exit.

⌋*(RS_EM_00103)*

[SWS_EM_02031] Application State Reporting ⌈ An `Application` shall report its state to the `Execution Management`, using the `ApplicationStateClient::ReportApplicationState()` interface. ⌋*(RS_EM_00101, RS_EM_00103)*

7.6.3 Machine State

`Machine State` defines the current set of running `Application`s. It is significantly influenced by vehicle-wide events and modes.

Each `Application` declares in its `Application Manifest` in which `Machine State`s it has to be active.

There are several mandatory machine states specified in this document that have to be present on each machine. Additional `Machine State`s can be defined on a machine specific basis and are therefore not standardized.

7.6.3.1 Machine State Management

`Machine State Management` is the ability to control the `Machine State` during the runtime of an Adaptive AUTOSAR ECU. `Machine State Management` 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:

- Integrated in the Execution Management
- As separate Machine State Management Application

7.6.3.1.1 Integrated in the Execution Management

When integrated, to request Machine State change, all Applications interested in doing so, are using the MachineStateClient API (see Section 8.2.2) to communicate, based on IPC mechanisms, with the Execution Management.

An overview of the first Use-Case with the interaction of the Execution Management and Applications is shown in Figrue 7.5.

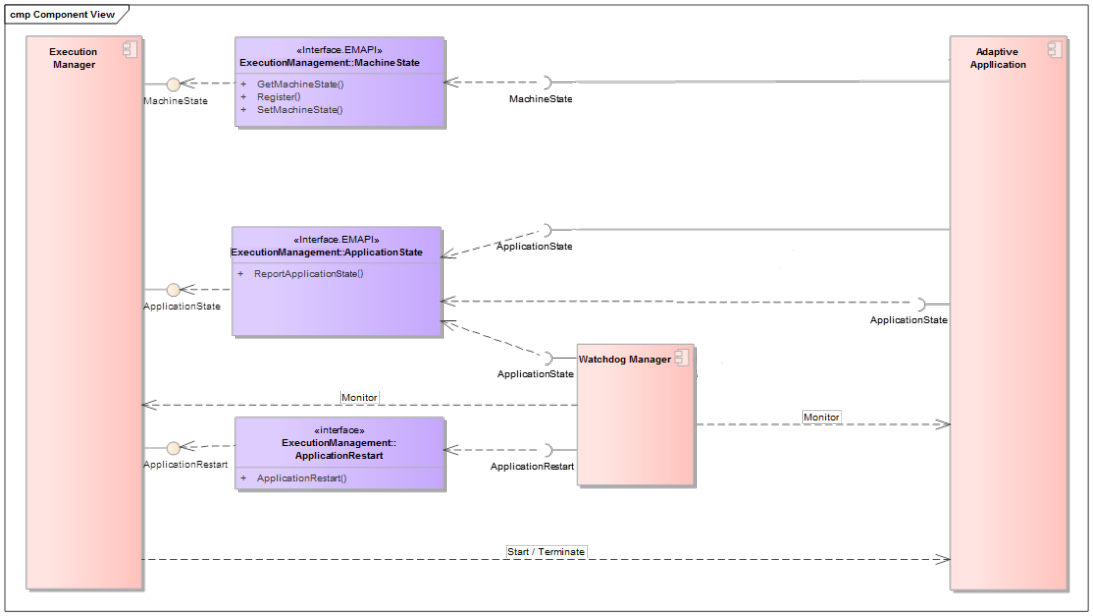

Figure 7.5: Machine State Management within Execution Management

The integrated approach has the disadvantage that the Execution Management is increasing in complexity as the algorithm and vendor specific knowledge is needed to implement it. Therefore a separation is also possible.

7.6.3.1.2 Separate Machine State Management Application

For this use-case, the Execution Management only provides operative mechanisms and interfaces to control the actual set of running Applications. The decision of state changes is fully given to the Application, that uses the MachineStateClient API and is then known as the Machine State Management Application.

[SWS_EM_01056] Machine State Management Application ⌈ It is recommended to have one Application within the Adaptive Platform that uses the MachineS-

tateClient API, and is responsible for the Machine State Management throughout using this API. ⌋(*RS_EM_00101*)

For this use-case, the MachineStateClient API doesn't contain any restrictions of Machine State change requests. This functionality has to be provided by the Machine State Management Application. The Machine State Management is here the central point where all Applications refer to, for retrieving the current and requesting new Machine States.

[SWS_EM_01057] Machine State Change arbitration ⌈ The Machine State Management Application shall arbitrate all Machine State change requests and request the arbitrated change at the Execution Management via the MachineStateClient::SetMachineState() API. ⌋(*RS_EM_00101*)

[SWS_EM_01027] Rejection of Client Requests ⌈ The Execution Management shall reject requests from any MachineStateClient::Register if there is already a MachineStateClient connected to the Execution Management. ⌋ (*RS_EM_00101*)

An overview of the second Use-Case with the interaction of the Machine State Management Application, the Execution Management and Applications is shown in Figrue 7.6.

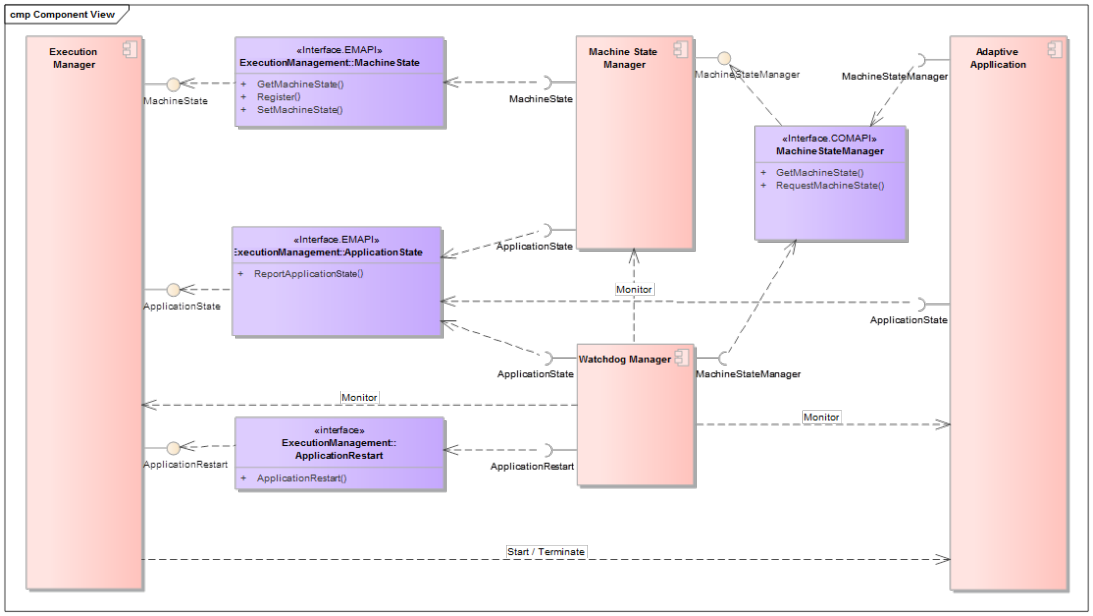

Figure 7.6: Machine State Management Application

The ara::com interfaces between Applications and the Machine State Management for requesting and retrieving Machine State is not specified and is therefore machine specific.

7.6.3.2 Startup

[SWS_EM_01023] `Machine State Startup` ⌈ The `Startup Machine State` shall be the first state to be active after the startup of `Execution Management Application`. Therefore, a `ModeDeclaration` for the `Startup` has to be defined in the `Machine Manifest`. ⌋*(RS_EM_00101)*

[SWS_EM_01037] `Machine State Startup` behavior ⌈ The following behavior apply for the `Startup Machine State`:

- All platform-level `Application`s shall be started, based on the reference from the `Application`'s `Process`es to the `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with the instanceRef to the `ModeDeclaration` in the role `ModeDependentStartupConfig.modeDeclaration` that belongs to the `Startup Machine State`.
- For startup of `Application`s, the startup requirements of section 7.4 apply.
- `Execution Management` shall wait for all started `Application`s until their `Application State Running` is reported.
- If that is the case, `Execution Management` shall notify the `MachineStateClient` that the `Startup Machine State` is ready to be changed.
- `Execution Management` shall not change the `Machine State` by itself until a new state is requested via the `MachineStateClient` API.

⌋*(RS_EM_00101)*

7.6.3.3 Shutdown

[SWS_EM_01024] `Machine State Shutdown` ⌈ The `Shutdown Machine State` shall be active after the `Shutdown Machine State` is requested via the `MachineStateClient` API. Therefore, a `ModeDeclaration` for the `Shutdown` has to be defined in the `Machine Manifest`. ⌋*(RS_EM_00101)*

[SWS_EM_01036] `Machine State Shutdown` behavior ⌈ The following behavior apply for the `Shutdown Machine State`:

- All `Application`s, including the platform-level `Application`s, that have a `Process State` different than `Idle` or `Terminated` shall be shutdown.
- For shutdown of `Application`s, the shutdown requirements of section 7.4 apply.
- When `Process State` of all `Application`s is `Idle` or `Terminated`, all `Application`s shall be started, based on the reference from the `Application`'s `Process`es to the `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with the instanceRef to the `ModeDeclaration`

in the role ModeDependentStartupConfig.modeDeclaration that belongs to the Shutdown Machine State.

⌋(*RS_EM_00101*)

[SWS_EM_01058] Shutdown of the Operating System ⌈ There shall be at least one Application consisting of at least one Process that has a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the Shutdown Machine State. This Application shall contain the actual mechanism(s) to initiate shutdown of the Operating System. ⌋(*RS_EM_00101*)

7.6.3.4 Restart

[SWS_EM_01025] Machine State Restart ⌈ The Restart Machine State shall be active after the Restart Machine State is requested via the MachineStateClient API. Therefore, a ModeDeclaration for the Restart has to be defined in the Machine Manifest. ⌋(*RS_EM_00101*)

[SWS_EM_01035] Machine State Restart behavior ⌈ The following behavior applies for the Restart Machine State:

- All Applications, including the platform-level Applications, that have a Process State different than Idle or Terminated shall be shutdown.
- For shutdown of Applications, the shutdown requirements of Section 7.4 apply.
- When Process State of all Applications is Idle or Terminated, all Applications shall be started, based on the reference from the Application's Processes to the ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the Restart Machine State.

⌋(*RS_EM_00101*)

[SWS_EM_01059] Restart of the Operating System ⌈ There shall be at least one Application consisting of at least one Process that has a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the Restart Machine State. This Application shall contain the actual mechanism(s) to initiate restart of the Operating System. ⌋(*RS_EM_00101*)

7.6.3.5 State Change

[SWS_EM_01026] **`Machine State` change** ⌈ A request of a `Machine State` change at the `Execution Management` via the `ApplicationStateClient::SetMachineState()` API shall lead to a state transition and hereof a state change to the requested `Machine State` in the `Execution Management`. ⌋*(RS_EM_00101)*

[SWS_EM_01060] **`Machine State` change behavior** ⌈ The following behavior applies for the transition of a current `Machine State` (referred to as *CurrentState*) to a requested `Machine State` (referred to as *RequestedState*):

1. For each `Application`: For each `Executable` of that `Application`, that has

   [

   - exactly one aggregation from the `Executable`'s `Process` containing a `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with an instanceRef to a `ModeDeclaration` in the role `ModeDependentStartupConfig.modeDeclaration` that belongs to the *CurrentState*

     and

   - no existing aggregation from the `Executable`'s `Process` containing a `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with an instanceRef to a `ModeDeclaration` in the role `ModeDependentStartupConfig.modeDeclaration` that belongs to the *RequestedState*

     and

   - a `Process State` different than [`Idle` or `Terminated`]

   ] or [

   - exactly one aggregation from the `Executable`'s `Process` containing a `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with an instanceRef to the `ModeDeclaration` in the role `ModeDependentStartupConfig.modeDeclaration` that belongs to the *CurrentState*

     and

   - exactly one aggregation from the `Executable`'s `Process` containing a `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with an instanceRef to a `ModeDeclaration` in the role `ModeDependentStartupConfig.modeDeclaration` that belongs to the *RequestedState*

     and

- different aggregated StartupOptions in the role StartupConfig.startupOption, referenced by the ModeDependentStartupConfigs in the role ModeDependentStartupConfig.startupConfig
  - with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *CurrentState*

    and
  - and with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *RequestedState*.

],

the Executable shall be shutdown. For shutdown the requirements of section 7.4 apply.

2. Wait until Process State of all affected Processes is Idle or Terminated.
3. For each Application: For each Executable of that Application, that has

[

- no existing aggregation from the Executable's Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to a ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *CurrentState*

  and
- exactly one aggregation from the Executable's Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to a ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *RequestedState*

  and
- a Process State that is [Idle or Terminated]

] or [

- exactly one aggregation from the Executable's Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *CurrentState*

  and

- exactly one aggregation from the Executable's Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to a ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *RequestedState*

  and

- different aggregated StartupOptions in the role StartupConfig.startupOption, referenced by the ModeDependentStartupConfigs in the role ModeDependentStartupConfig.startupConfig
  - with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *CurrentState*

    and
  - and with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.modeDeclaration that belongs to the *RequestedState*.

⌋,

the Executable shall be started. For startup the requirements of section 7.4 apply.

4. Wait until Process State of all affected Processes is Running.
5. When Machine State change is originated by the MachineStateClient API, a confirmation of the change shall be sent to the ApplicationStateClient.

⌋*(RS_EM_00101)*

[SWS_EM_01028] GetMachineState API ⌈ The Execution Management shall provide the interface MachineStateClient::GetMachineState() to retrieve the current Machine State. ⌋*(RS_EM_00101)*

[SWS_EM_01029] SetMachineState API ⌈ The Execution Management shall provide the interface MachineStateClient::SetMachineState() to request a change to a new Machine State. ⌋*(RS_EM_00101)*

[SWS_EM_01034] Deny SetMachineState API Request ⌈ The Execution Management shall deny Machine State change requests, that are received before confirmation of the previous Machine State change. If a request is denied, Execution Management shall return an error code to the requester (see 8.1.2). ⌋*(RS_EM_00101)*

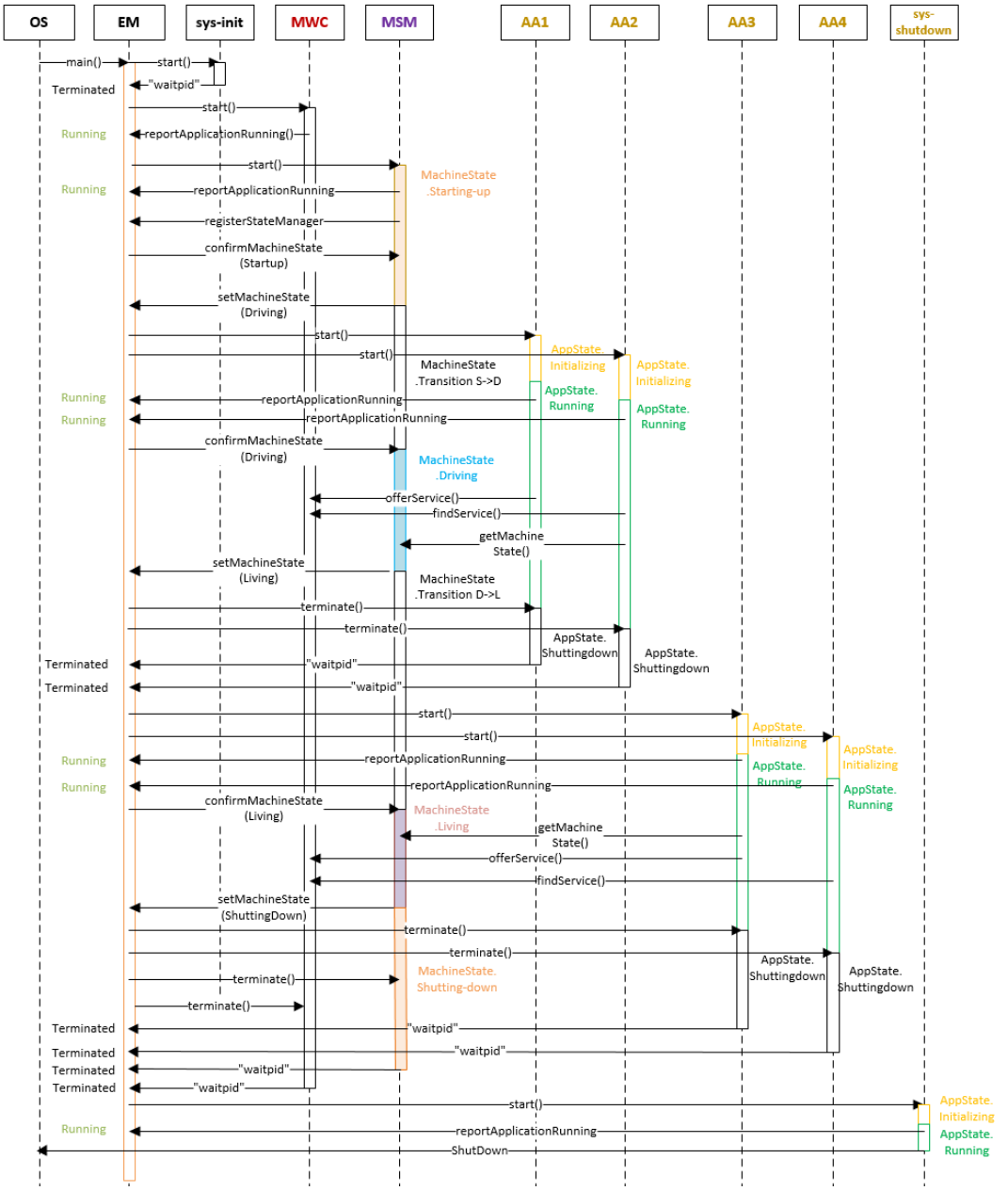

Figure 7.7: Adaptive Platform - Lifecycle

8 API specification

8.1 Type definitions

8.1.1 ApplicationState

Name:	**[SWS_EM_02000]** ⌈`ApplicationState`⌋*(RS_EM_00103)*		
Type:	`Scoped Enumeration of uint8_t`		

Range:	`kInitializing`
	`kRunning`
	`kShuttingdown`
		`0`
		`1`
		`2`
			`--`
			`--`
			`--`

Syntax:	`enum class ApplicationState : uint8_t {`
	`kInitializing,`
	`kRunning,`
	`kShuttingdown`
	`};`		

Header file:	application_state_client.hpp		
Description:	Defines the states of an `Application` (see 7.6.2).		

Table 8.1: ApplicationState

8.1.2 StateError

Name:	**[SWS_EM_02005]** ⌈`StateError`⌋*(RS_EM_00101)*		
Type:	`Scoped Enumeration of uint8_t`		

Range:	`kSuccess`
	`kInvalidState`
	`kInvalidRequest`
	`kTimeout`
		`0`
		`1`
		`2`
		`3`
			`--`
			`--`
			`--`
			`--`

Syntax:	`enum class StateError : uint8_t {`
	`kSuccess,`
	`kInvalidState,`
	`kInvalidRequest`
	`kTimeout`
	`};`		

Header file:	state_error.hpp		
Description:	Defines the error codes for `Machine State` operations.		

Table 8.2: StateError

8.2 Class definitions

8.2.1 ApplicationStateClient class

The Application State API provides the functionality for an `Application` to report its state to the `Execution Management`.

[SWS_EM_02001] ⌈ The `ApplicationStateClient` class shall be defined in the `application_state_client.hpp` header file. ⌋*(RS_EM_00103)*

8.2.1.1 ApplicationStateClient::ApplicationStateClient

Service name:	**[SWS_EM_02030]** ⌈ApplicationStateClient Constructor⌋ *(RS_EM_00103)*	
Syntax:	`ApplicationStateClient();`	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	OpenPipeException	In case the underlying POSIX FIFO `open()` operation fails.
Description:	Creates an instance of ApplicationStateClient which opens the `Execution Management`'s communication channel (e.g. POSIX FIFO available under: "/usr/run/execution-manager/appstate-server-fifo") for reporting the application state. Each `Application` shall create an instance of this class to report its state.	

Table 8.3: ApplicationStateClient Constructor

8.2.1.2 ApplicationStateClient::~ApplicationStateClient

Service name:	**[SWS_EM_02002]** ⌈ApplicationStateClient Destructor⌋ *(RS_EM_00103)*
Syntax:	`~ApplicationStateClient();`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	~ApplicationStateClient deletes the ApplicationStateClient instance.

Table 8.4: ApplicationStateClient Destructor

8.2.1.3 ApplicationStateClient::ReportApplicationState

Service name:	**[SWS_EM_02003]** ⌈ApplicationStateClient::ReportApplicationState⌋ *(RS_EM_00103)*	
Syntax:	`void ReportApplicationState( ApplicationState state );`	
Parameters (in):	state	Value of the `Application`s state
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	None	
Description:	Interface for an `Application` to report the state to `Execution Management`. It sends the state value via underlying IPC mechanism (e.g. POSIX FIFO) to the `Execution Management`.	

Table 8.5: ApplicationStateClient::ReportApplicationState

8.2.2 MachineStateClient class

The `MachineStateClient` class provides the functionality for an `Application` to request a `Machine State` switch or to retrieve the current Machine State to/from `Execution Management`.

The `Application` responsible for managing and controlling the states, `Machine State Management` has to instantiate this class. The `Execution Management` allows only one client to be registered (See [SWS_EM_01027]).

[SWS_EM_02006] ⌈ The `MachineStateClient` class shall be defined in the `machine_state_client.hpp` header file. ⌋*(RS_EM_00101)*

8.2.2.1 MachineStateClient::MachineStateClient

Service name:	**[SWS_EM_02007]** ⌈MachineStateClient Constructor⌋*(RS_EM_00101)*	
Syntax:	`MachineStateClient( std::string path );`	
Parameters (in):	path	Path to the platform-wide known communication channel.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	OpenPipeException	In case the underlying POSIX FIFO `open()` operation fails.
Description:	Creates an instance of MachineStateClient which opens the `Execution Management`'s communication channel (e.g. POSIX FIFO available under: "/usr/run/execution-manager/machinestate-server-fifo") for retrieving or requesting `Machine State`s from/to `Execution Management`.	

Table 8.6: MachineStateClient Constructor

8.2.2.2 MachineStateClient::~MachineStateClient

Service name:	**[SWS_EM_02008]** ⌈MachineStateClient Destructor⌋*(RS_EM_00101)*
Syntax:	`~MachineStateClient();`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	~MachineStateClient deletes the MachineStateClient instance.

Table 8.7: MachineStateClient Destructor

8.2.2.3 MachineStateClient::Register

Service name:	**[SWS_EM_02009]** ⌈MachineStateClient::Register⌋*(RS_EM_00101)*

Syntax:	`StateError Register( std::string app_name, uint32_t timeout );`	
Parameters (in):	app_name	Name of the `Application` to be registered to `Execution Management`
	timeout	Time to wait in milliseconds to get the confirmation of registration from `Execution Management`.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	kSuccess	Registration to `Execution Management` succeeded and confirmed.
	kTimeout	No registration confirmation received from `Execution Management`.
Exceptions:	None	
Description:	Register the MachineStateClient to the `Execution Management`. That is done by the `Machine State Manager Application`.	

Table 8.8: MachineStateClient::Register

8.2.2.4 MachineStateClient::GetMachineState

Service name:	**[SWS_EM_02014]** ⌈MachineStateClient::GetMachineState⌋ *(RS_EM_00101)*	
Syntax:	`StateError GetMachineState( uint32_t timeout, std::string &state );`	
Parameters (in):	timeout	Time to wait in milliseconds to get the required information back from `Execution Management`.
Parameters (inout):	None	
Parameters (out):	state	String containing the current `Machine State`. Empty string if return value is kTimeout.
Return value:	kSuccess	Retrieval operation succeeded.
	kTimeout	No answer received from `Execution Management`.
Exceptions:	None	
Description:	Retrieve the current `Machine State` from the `Execution Management`. It sends a request over the corresponding IPC channel (e.g POSIX FIFO) of `Execution Management`.	

Table 8.9: MachineStateClient::GetMachineState

8.2.2.5 MachineStateClient::SetMachineState

Service name:	**[SWS_EM_02019]** ⌈MachineStateClient::SetMachineState⌋ *(RS_EM_00101)*	
Syntax:	`StateError SetMachineState( std::string state, uint32_t timeout );`	
Parameters (in):	state	String containing the requested `Machine State`.

	timeout	Time to wait in milliseconds to get `Machine State` change confirmation back form `Execution Management`.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	kSuccess	`Machine State` change operation succeeded.
	kTimeout	No confirmation received from `Execution Management`.
	kInvalidState	Invalid state provided.
	kInvalidRequest	Request cannot be fulfilled as it was received before confirmation of the previous change.
Exceptions:	None	
Description:	Requests a new `Machine State` at the `Execution Management` via the corresponding IPC channel (e.g POSIX FIFO) of `Execution Management` for changing the current `Machine State`. The method returns after `Machine State` change is confirmed, or on timeout.	

Table 8.10: MachineStateClient::SetMachineState

A Not applicable requirements

[SWS_EM_99999] ⌈ These requirements are not applicable as they are not within the scope of the 2017-03 release. ⌋*(RS_EM_00003, RS_EM_00004, RS_EM_00005, RS_EM_00006, RS_EM_00007, RS_EM_00008, RS_EM_00013, RS_EM_00050, RS_EM_00051, RS_EM_00052)*

B Mentioned Class Tables

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document but which are not contained directly in the scope of describing specific meta-model semantics.

Enumeration	**CommandLineOptionKindEnum**
Package	M2::AUTOSARTemplates::AdaptivePlatform::Process
Note	This enum defines the different styles how the command line option appear in the command line. **Tags:** atp.Status=draft
Literal	***Description***
commandLineLongForm	Long form of command line option. **Tags:** atp.EnumerationValue=1
commandLineShortForm	Short form of command line option. **Tags:** atp.EnumerationValue=0
commandLineSimpleForm	In this case the command line option does not have any formal structure. Just the value is passed to the program. **Tags:** atp.EnumerationValue=2

Table B.1: CommandLineOptionKindEnum

Class	**ModeDeclaration**			
Package	M2::AUTOSARTemplates::CommonStructure::ModeDeclaration			
Note	Declaration of one Mode. The name and semantics of a specific mode is not defined in the meta-model.			
Base	ARObject, AtpClassifier, AtpFeature, AtpStructureElement, Identifiable, MultilanguageReferrable, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
value	PositiveInteger	0..1	attr	The RTE shall take the value of this attribute for generating the source code representation of this ModeDeclaration.

Table B.2: ModeDeclaration

Class	**ModeDependentStartupConfig**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Process			
Note	This meta-class defines the startup configuration for the process depending on a collection of machine states. **Tags:** atp.Status=draft			
Base	ARObject			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
executionDependency	ExecutionDependency	*	aggr	This attribute defines that all processes that are referenced via the ExecutionDependency shall be launched and shall reach a certain ApplicationState before the referencing process is started. **Tags:** atp.Status=draft
machineMode	ModeDeclaration	*	iref	This represent the applicable modeDeclaration. **Tags:** atp.Status=draft
startupConfig	StartupConfig	1	ref	Reference to a reusable startup configuration with startup parameters. **Tags:** atp.Status=draft

Table B.3: ModeDependentStartupConfig

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Process			
Note	This meta-class provides information required to execute the referenced executable. **Tags:** atp.Status=draft; atp.recommendedPackage=Processes			
Base	ARElement, ARObject, AtpClassifier, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
applicationModeMachine	ModeDeclarationGroupPrototype	0..1	aggr	Set of ApplicationStates (Modes) that are defined for the process. **Tags:** atp.Status=draft
executable	Executable	0..1	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef **Tags:** atp.Status=draft
modeDependentStartupConfig	ModeDependentStartupConfig	*	aggr	Applicable startup configurations. **Tags:** atp.Status=draft

Table B.4: Process

Class	**StartupConfig**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Process			
Note	This meta-class represents a reusable startup configuration for processes.. **Tags:** atp.Status=draft			
Base	ARObject, Identifiable, MultilanguageReferrable, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
resourceGroup	ResourceGroup	*	ref	Reference to applicable resource groups. **Tags:** atp.Status=draft
schedulingPolicy	SchedulingPolicyKindEnum	0..1	attr	This attribute represents the ability to define the scheduling policy.
schedulingPriority	Integer	0..1	attr	This is the scheduling priority requested by the application itself.
startupOption	StartupOption	*	aggr	Applicable startup options **Tags:** atp.Status=draft

Table B.5: StartupConfig

Class	**StartupOption**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Process			
Note	This meta-class represents a single startup option consisting of option name and an optional argument. **Tags:** atp.Status=draft			
Base	ARObject			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
optionArgument	String	0..1	attr	This attribute defines option value.
optionKind	CommandLineOptionKindEnum	1	attr	This attribute specifies the style how the command line options appear in the command line.
optionName	String	0..1	attr	This attribute defines option name.

Table B.6: StartupOption