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	17-10

Document Change History			
Date	**Release**	**Changed by**	**Description**
2017-10-27	17-10	AUTOSAR Release Management	• State Management elaboration, introduction of Function Groups • Recovery actions for Platform Health Management • Resource limitation and deterministic execution
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 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	Dependencies between `Executable` instances can be configured to define a sequence for starting and terminating them.
Execution Management	The element of the `Adaptive Platform` responsible for the ordered startup and shutdown of the `Adaptive Platform` and the `Application`s.
State Manager	The element of the `Execution Management` defining modes of operation for `Adaptive Platform`. It allows flexible definition of functions which are active on the platform at any given time. Architecture and functionality of State Management are still under dicussion. A consolidated description will follow in a later release.
Machine State	The element of the `State Management` which characterize the current status of the machine. It defines a set of active `Application`s for any certain situation. The set of `Machine State`s is machine specific and it will be deployed in the `Machine Manifest`. `Machine State`s are mainly used to control machine lifecycle (startup/shut-down/restart) and platform-level processes.
Function Group State	The element of the `State Management` which characterize the current status of functionally coherent user-level `Application`s. The set of `Function Group`s and their `Function Group State`s is machine specific and it will be deployed in the `Machine Manifest`.
Time Determinism	The results of a calculation are guaranteed to be available before a given deadline.
Data Determinism	The results of a calculation only depend on the input data and are reproducible, assuming a given initial internal state.
Full Determinism	Combination of Time and Data Determinism.

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 Operating System Interface
AUTOSAR_RS_OperatingSystemInterface

[4] Requirements on Execution Management
AUTOSAR_RS_ExecutionManagement

[5] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology

[6] Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7
http://pubs.opengroup.org/onlinepubs/9699919799/

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:

- 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.
- Resource limitation and deterministic execution will be expanded with more properties and formal requirements (see 7.7 and 7.8).
- ECU/VM reset needs more clarification.
- Error handling and timeout is not finished and will be expanded.

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 [3]. 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, there are no other library dependencies.

6 Requirements tracing

The following tables reference the requirements specified in [4] 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_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 Executables at startup of Executable.	[SWS_EM_NA]
[RS_EM_00004]	The Execution Management shall support the authentication and authorization of Executables at startup of Executable	[SWS_EM_NA]
[RS_EM_00005]	The Execution Management shall support the configuration of OS resource budgets for Executable and groups of Executables	[SWS_EM_NA]
[RS_EM_00006]	The Execution Management shall support the supervision of available and required OS resource budgets for Executables and groups of Executables during installation	[SWS_EM_NA]
[RS_EM_00007]	The Execution Management shall support allocation of dedicated resources for the Executable (e.g GPU)	[SWS_EM_NA]
[RS_EM_00008]	The Execution Management shall support the binding of Executable threads to a specified set of processor cores.	[SWS_EM_01201]
[RS_EM_00009]	Only Execution Management shall start Executables	[SWS_EM_01030]
[RS_EM_00010]	The Execution Management shall support multiple instances of Executables	[SWS_EM_01012] [SWS_EM_01033]
[RS_EM_00011]	Execution Managemen shall support self-initiated graceful shutdown of Executable instances	[SWS_EM_01005]
[RS_EM_00012]	Application Manifes shall support unique identification of Executable instances	[SWS_EM_01017] [SWS_EM_01050]
[RS_EM_00013]	Execution Management shall support configurable recovery actions	[SWS_EM_01016] [SWS_EM_01018] [SWS_EM_01061] [SWS_EM_01062]

Requirement	Description	Satisfied by
[RS_EM_00050]	The Execution Managemen shall do a system-wide coordination of activities	[SWS_EM_NA]
[RS_EM_00051]	The Execution Management shall provide functions to the Executable for configuring external trigger conditions for its activities	[SWS_EM_NA]
[RS_EM_00052]	The Execution Management shall provide functions to the Executable for configuring cyclic triggering of its activities	[SWS_EM_NA]
[RS_EM_00053]	The Execution Management shall provide functions to support redundant execution of Executables	[SWS_EM_NA]
[RS_EM_00100]	The Execution Management shall support the ordered startup and shutdown of Executables	[SWS_EM_01000] [SWS_EM_01001] [SWS_EM_01050] [SWS_EM_01051]
[RS_EM_00101]	The Execution Management shall provide State Management functionality	[SWS_EM_01013] [SWS_EM_01023] [SWS_EM_01024] [SWS_EM_01025] [SWS_EM_01026] [SWS_EM_01028] [SWS_EM_01032] [SWS_EM_01034] [SWS_EM_01035] [SWS_EM_01036] [SWS_EM_01037] [SWS_EM_01056] [SWS_EM_01058] [SWS_EM_01059] [SWS_EM_01060] [SWS_EM_01107] [SWS_EM_01108] [SWS_EM_01109] [SWS_EM_01110] [SWS_EM_01111] [SWS_EM_01112] [SWS_EM_02005] [SWS_EM_02006] [SWS_EM_02007] [SWS_EM_02008] [SWS_EM_02031] [SWS_EM_02044] [SWS_EM_02047] [SWS_EM_02048] [SWS_EM_02049] [SWS_EM_02050] [SWS_EM_02051] [SWS_EM_02054] [SWS_EM_02055] [SWS_EM_02056] [SWS_EM_02057] [SWS_EM_02070] [SWS_EM_02072] [SWS_EM_02073] [SWS_EM_02074] [SWS_EM_02075]
[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_01053] [SWS_EM_01055] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02030] [SWS_EM_02031] [SWS_EM_02071]
[RS_EM_00110]	Execution Management shall support diagnostic reset cause	[SWS_EM_02041] [SWS_EM_02042] [SWS_EM_02043]

7 Functional specification

7.1 Technical Overview

This chapter presents a short summary of the relationship between Application, Executable, and Process.

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.

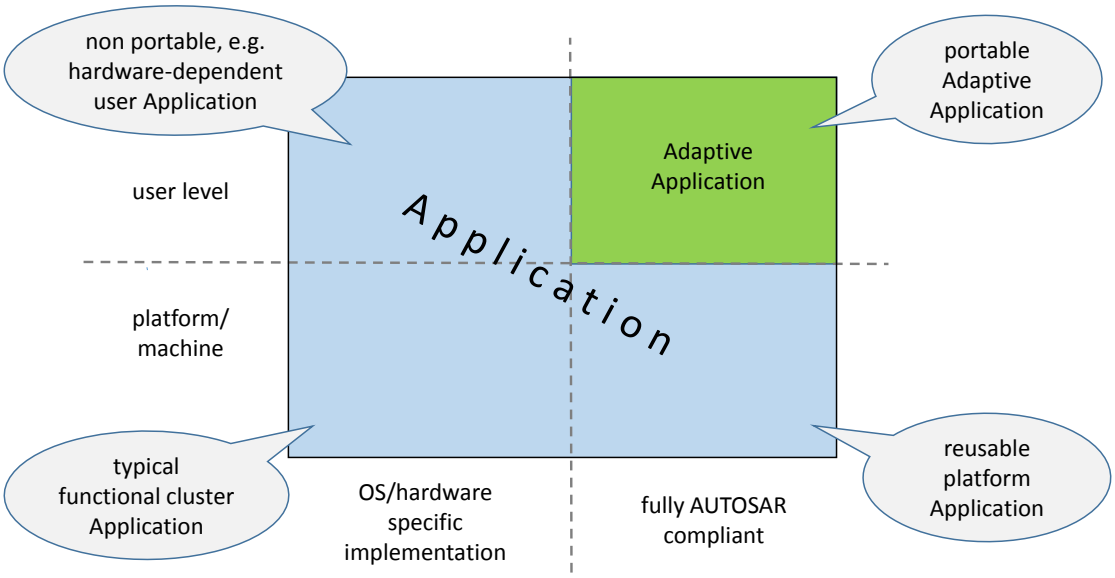

Figure 7.1: Types of Applications

An `Adaptive Application` is the result of functional development and is the unit of delivery for `Machine` specific configuration and integration. 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].

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 `Executable`s.

The lifecycle of `Executable`s usually consists of:

Process Step	Software	Meta Information
Development and 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], and `Service Instance Manifest` (not used by Execution Management).
Deployment and Removal	Binary installed on the target `Machine`.	Processed Manifests, 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 Manifests to start up and configure each process individually.

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.

Figure 7.2 shows the lifecycle of an Executable from deployment to execution.

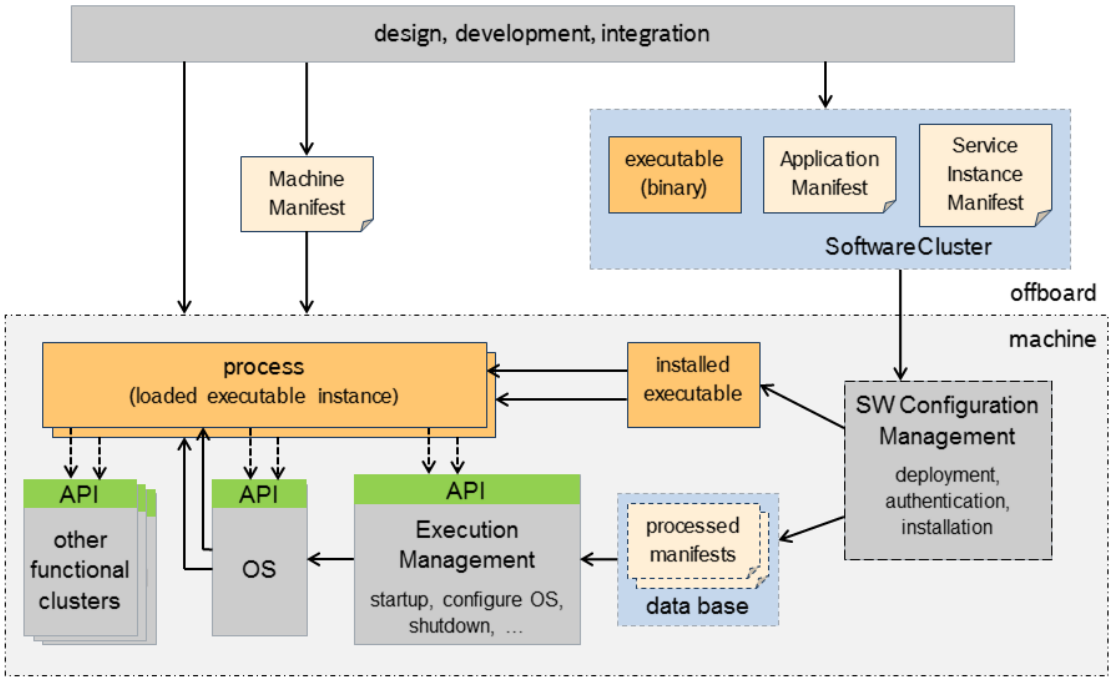

Figure 7.2: Executable Lifecycle from deployment to execution

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

A Process is a started instance of an Executable. For details on how Execution Management starts and stops Processes see 7.4.

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 7.5).

7.1.5 Application Manifest

The Application Manifest consists of parts of the Application design information which is provided by the application developer in an application description, and additional machine-specific information which is added at integration time. For details on the Application Manifest contents see chapter 7.9. A formal specification can be found in [2].

An Application Manifest is created together with a Service Instance Manifest (not used by Execution Management) at integration time and deployed onto a Machine together with the Executable it is attached to. It describes in a standardized way the machine-specific configuration of Process properties (startup parameters, resource group assignment, 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 or per Function Group State (see 7.5 and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01014], [TPS_MANI_01015], [TPS_MANI_01059], [TPS_MANI_01017] and [TPS_MANI_01041]).

7.1.6 Machine Manifest

The Machine Manifest is also created at integration time for a specific Machine and is deployed like Application Manifests 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 or a Service Instance Manifest.

The contents of a Machine Manifest includes the configuration of Machine properties and features (resources, safety, security, etc.), e.g. configured Machine States and Function Group States, resource groups, access right groups, scheduler configuration, SOME/IP configuration, memory segmentation. For details see [2].

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 when, and possibly in which order, to start or stop the deployed Applications, based on information in the Machine Manifest and Application Manifests.

   Depending on the Machine State or on a Function Group 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 derives an ordering for startup/shutdown within the State Management framework, based on declared Application dependencies. The dependencies are described in the Application Manifests, see [TPS_MANI_01041].

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 Manager as part of 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 Manager is running no other element of Adaptive Platform initiates Application execution.

Platform monitoring – the Execution Management can perform Application monitoring, also in conjunction with the Platform Health Management. Particular design aspects of resource monitoring are shown in chapter 7.7.

Platform shutdown – the Execution Manager 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 States* 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 Applications 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 Applications process after it has been scheduled and it has reported Running State to the Execution Manager. ⌋*(RS_EM_00103)*

[SWS_EM_01005] Terminating Process State ⌈ The **Terminating** Process State shall apply either after an Applications process has received the termination indication from the Execution Manager or after it has decided to self-terminate and informed the Execution Manager. ⌋*(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 Applications process performs persistent storage of the working data, frees all Applications process internal resources, and exits.

[SWS_EM_01006] Terminated Process State ⌈ The **Terminated** Process State shall apply after the Applications process has been terminated and the process resources have been freed. For that, Execution Manager shall observe the exit status of all Applications 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 Applications 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 Applications process is supposed to run only once before any other application execution. ⌋*(RS_EM_00103)*

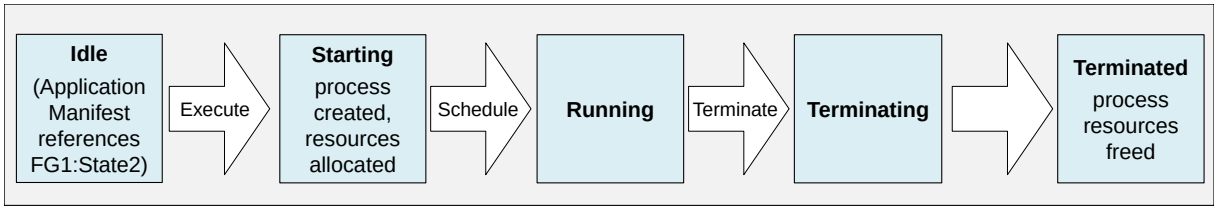

Figure 7.3: Process Lifecycle

7.4.2 Startup and shutdown

[SWS_EM_01050] Start dependent Application Executables ⌈ The Execution Management shall respect Application Dependencys and start any Application Executables in this list first. In case no dependency is specified between two Application Executables, they can be started in an arbitrary order. ⌋ *(RS_EM_00012, RS_EM_00100)*

[SWS_EM_01051] Shutdown Application Executables ⌈ The Execution Management shall respect Application Dependencys and shutdown dependent Application Executables 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 or 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 Manager 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 user-level Applications.

[SWS_EM_01000] Startup order ⌈ The startup order of the platform-level Applications is determined by the Execution Management, based on Machine Manifest and Application Manifest information. ⌋(*RS_EM_00100*) Please see Section 7.9.1.

Figure 7.4 shows the overall startup sequence.

[1] Typically the *init* process

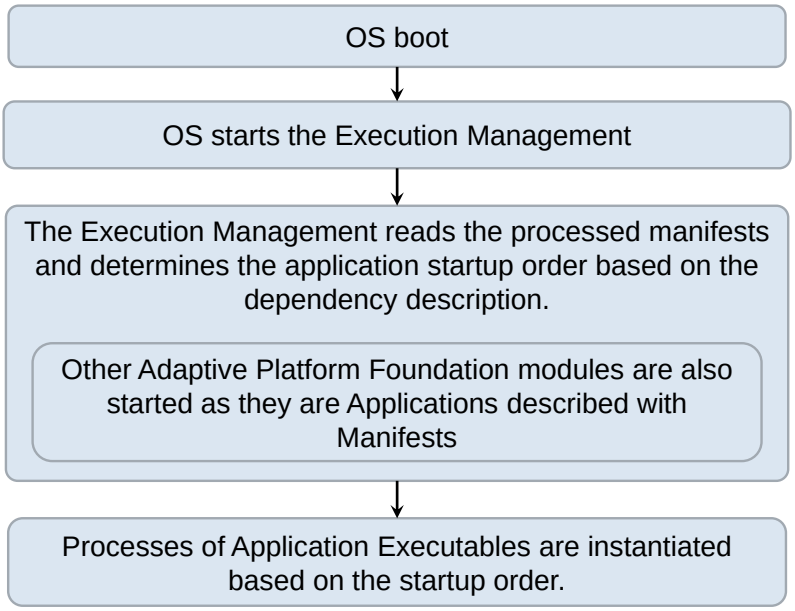

Figure 7.4: Startup sequence

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. In this release, this only applies to platform-level Applications at machine startup and shutdown, see [constr_1484] in [2].

The startup 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.

User-level applications use service discovery mechanisms of the Communication Management and should not depend on startup dependencies. Which Executable instances are running depends on the current Machine State and on the current Function Group States, see 7.5. The integrator must ensure that all service dependencies are mapped to State Management configuration, i.e. that all dependent Executable instances are running when needed.

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 7.5.2.

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 B's 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 B's 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>
```

Processes are only started by the Execution Manager if they reference a requested Machine State or Function Group State, but not because of configured Execution Dependencies. Execution Dependencies are only used to control a startup or terminate sequence at state transitions or at machine startup/shutdown.

[SWS_EM_01001] Execution Dependency error ⌈ If an Execution Dependency is configured in a ModeDependentStartupConfig of a starting or already running process which references a process that is not already in the *Running Application State* or being started at a Machine State or Function Group State transition (simple dependency), or that is not in the *Terminated Application State* (One-Shot Application dependency), or if two or more Applications have mutual dependencies, this shall be considered to be a configuration error. ⌋*(RS_EM_00100)*

Example: Let's assume we have a process "A" that depends on the *Running Application State* of a process "B". At a Machine State transition, process "A" shall be started, because it references the new Machine State. However, process "B" does not reference that Machine State, so it is not started. Due to the Execution Depen-

dency between the two processes, process “A” would never start running in the new Machine State because it waits forever for process “B”, which shall be considered a configuration error.

7.5 State Management

7.5.1 Overview

State Management provides a mechanism to define the state of the operation for an Adaptive Platform. The Application Manifest allows definition in which states the Application Executable instances have to run (see [2]). 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 actually needed.

Four different states are relevant for Execution Management:

- Application State, see 7.5.2
- Process State

  Process States are managed by an Execution Management internal state machine. For details see Section 7.4.1.
- Machine State, see 7.5.3
- Function Group State, see 7.5.4

An example for the interaction between these states will be shown in section 7.5.5.2.

7.5.2 Application State

The *Application State* characterizes the internal lifecycle of any instance of an Application Executable. The states are defined by the ApplicationState enumeration.

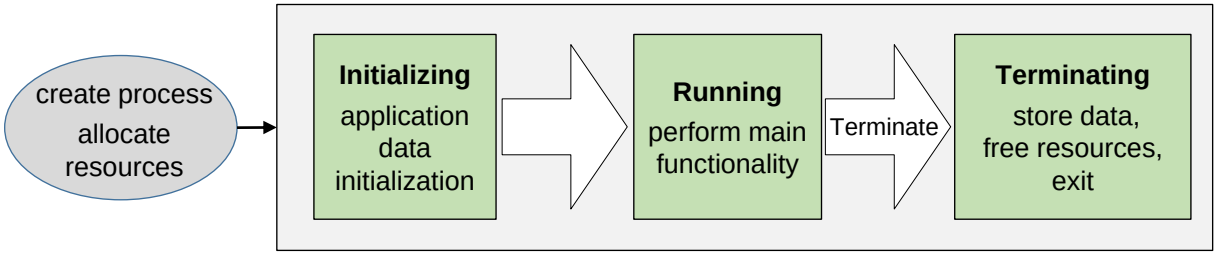

Figure 7.5: Application States

[SWS_EM_01053] Application State Running ⌈ Once the initialization of an Application Executable instance is complete, it shall switch to the Running state by setting the state to kRunning. ⌋*(RS_EM_00103)*

[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 Executable instance shall switch to the Terminating state and update it's state to the kTerminating enumeration value.
- During the Terminating state, the Application shall free internally used resources.
- When the Terminating state finishes, the Application Executable instance shall exit.

⌋*(RS_EM_00103)*

[SWS_EM_02031] Application State Reporting ⌈ An Application Executable instance shall report its state to the Execution Management using the ApplicationClient::ReportApplicationState interface. It has to be reported immediately after it has been changed. ⌋*(RS_EM_00101, RS_EM_00103)*

7.5.3 Machine State

Requesting and reaching a Machine State is, besides using Function Group States (see 7.5.4), one way to define the current set of running Application Executable instances. It is significantly influenced by vehicle-wide events and modes.

Each Application can declare in its Application Manifest in which Machine States 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 States can be defined on a machine specific basis and are therefore not standardized.

[SWS_EM_01032] Machine States ⌈ A ModeDeclaration for each required Machine State has to be defined in the Machine Manifest. The Execution Manager shall obtain the Machine States from the Machine Manifest. ⌋ *(RS_EM_00101)*

The Machine States are determined and requested by the State Manager, see 7.5.5.1. For details on state change management see 7.5.6.

7.5.3.1 Startup

[SWS_EM_01023] Machine State Startup ⌈ The Startup Machine State shall be the first state to be active after the startup of Execution Manager. 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 Applications configured for Startup shall be started. Applications configured for Startup are based on the reference from the Applications Processes to the ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the Startup Machine State.
- For startup of Applications, the startup requirements of section 7.4 apply.
- The Execution Manager shall wait for all started Applications until their Application State Running is reported.
- If that is the case, the Execution Manager shall notify the State Manager that the Startup Machine State is ready to be changed.
- The Execution Manager shall not change the Machine State by itself until a new state is requested by the State Manager.

⌋*(RS_EM_00101)*

7.5.3.2 Shutdown

[SWS_EM_01024] Machine State Shutdown ⌈ The Shutdown Machine State shall be active after the Shutdown Machine State is requested by the State Manager. 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 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 configured for Shutdown shall be started. Applications configured for Shutdown are based on the reference from the Applications Processes to the ModeDependentStartupConfig in the role Process.modeDe-

pendentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode 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.machineMode 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*)

7.5.3.3 Restart

[SWS_EM_01025] Machine State Restart ⌈ The Restart Machine State shall be active after the Restart Machine State is requested by the State Manager. 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 configured for Restart shall be started. Applications configured for Restart are based on the reference from the Applications Processes to the ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode 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.machineMode 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*)

7.5.4 Function Group State

If more than one group of functionally coherent Applications is installed on the same machine, the Machine State mechanism is not flexible enough to control these functional clusters individually, in particular if they have to be started and terminated with interleaving lifecycles. Many different Machine States would be required in this case to cover all possible combinations of active functional clusters.

To support this use case, Function Group States can be configured in addition to Machine States. Other use cases where starting and terminating individual groups of processes might be necessary include diagnostics and error recovery.

In general, Machine States are used to control machine lifecycle (startup/shutdown/restart), platform level processes, and other infrastructure, while Function Group States individually control groups of functionally coherent user level Application processes.

Figure 7.6 shows an example state change sequence where several processes reference Machine States and Function Group States of two Function Groups **FG1** and **FG2**. For simplicity, only the three static Process States Idle, Running, and Terminated are shown for each process.

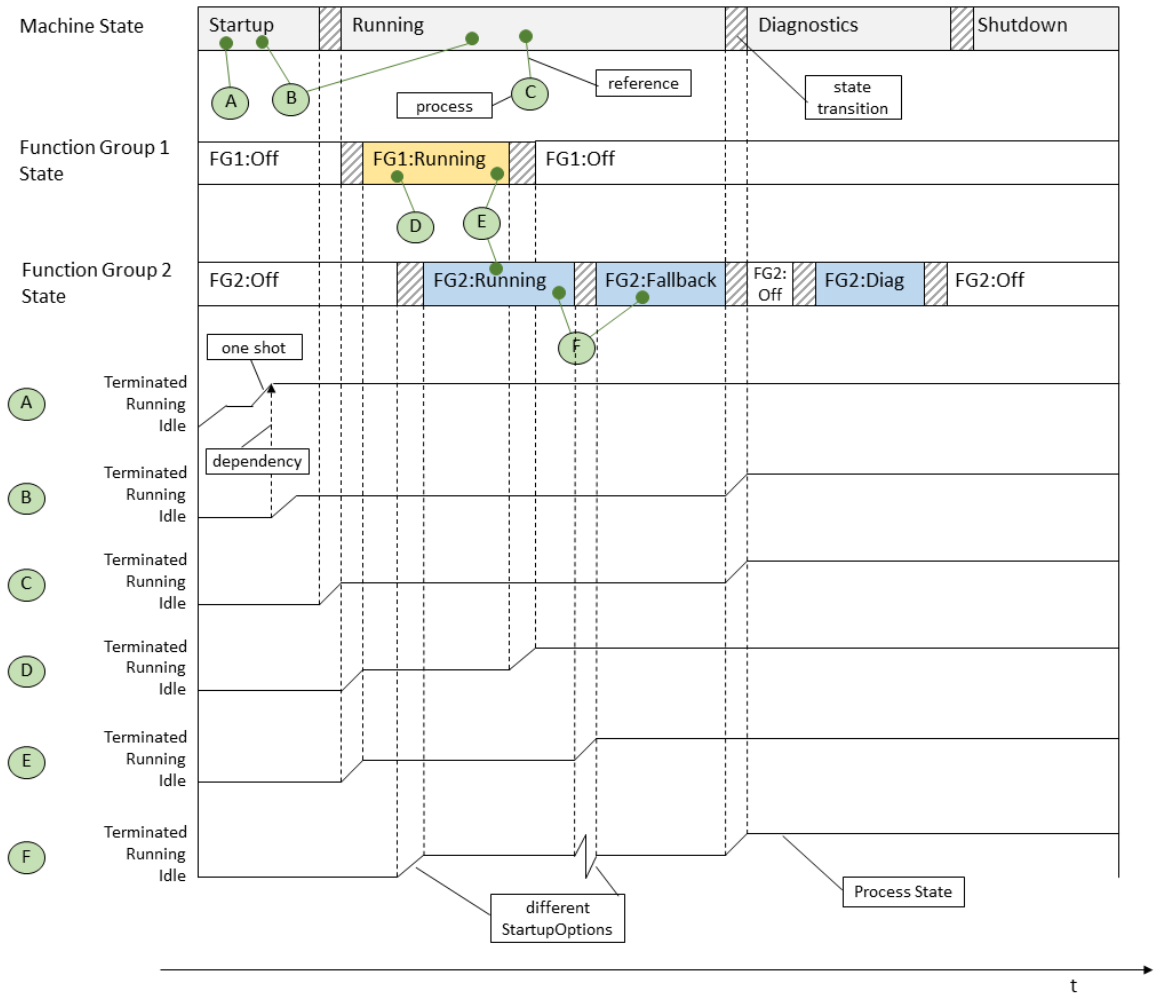

Figure 7.6: State dependent process control

- Process **A** references the `Machine State` `Startup`. It is a one shot process, i.e. it terminates after executing once.
- Process **B** references `Machine State`s `Startup` and `Running`. It depends on the termination of Process **A**, i.e. an Application Dependency has been configured, as described in 7.4.3.1
- Process **C** references `Machine State` `Running` only. It terminates when `Machine State` `Diagnostics` is requested by the `State Manager`.
- Process **D** references `Function Group State` `FG1:Running` only.
- Process **E** references `FG1:Running` and `FG2:Running`. Because it references states of different `Function Group`s, it must use the same startup configuration (`StartupConfig`) in all states to avoid sequence dependent behaviour.

- Process **F** references FG2:Running and FG2:Fallback. It has different startup configurations assigned to the two states, therefore it terminates at the state transition and starts again, using a different startup configuration.

System design and integration must ensure that enough resources are available on the machine at any time, i.e. the added resource consumption of all processes which reference simultaneously active States must be considered.

The Function Group States are determined and requested by the State Manager, see 7.5.5.1. For details on state change management see 7.5.6.

[SWS_EM_01107] Function Group name ⌈ A unique name for each Function Group has to be defined in the Machine Manifest. The Execution Manager shall obtain the name of the Function Group from the Machine Manifest to set-up the Function Group specific state management. ⌋*(RS_EM_00101)*

[SWS_EM_01108] Function Group State ⌈ A ModeDeclaration for each required Function Group State has to be defined in the Machine Manifest. Each Function Group State must be assignable to a specific Function Group. The Execution Manager shall obtain the Function Group States from the Machine Manifest. ⌋*(RS_EM_00101)*

[SWS_EM_01109] State References ⌈ Each instance of an Application Executable shall reference in its Application Manifest one or more Function Group States of the same or of different Function Groups and/or one or several Machine States. ⌋*(RS_EM_00101)* In the event of a misconfigured system, the Execution Manager shall not start an instance which does not reference at least one State.

[SWS_EM_01110] Off States ⌈ Each Function Group has an Off State which shall be used as default Function Group State, if no other state is requested. ⌋*(RS_EM_00101)*

[SWS_EM_01111] No reference to Off State ⌈ The Off Function Group States shall not be referenced in any Application Manifest. ⌋*(RS_EM_00101)*

[SWS_EM_01112] StartupConfig ⌈ Application Executable instances reference in their Application Manifest the states in which they want to be executed. A state can be a Function Group State or a Machine State. If an Application Executable instance references Function Group States which belong to more than one Function Group, or if it references both Machine States and Function Group States, then only one startup configuration (StartupConfig) shall be configured, which is then valid for all referenced states. ⌋*(RS_EM_00101)*

This restriction prevents undefined behaviour, because if an Application Executable instance references states of different Function Groups, which can be active simultaneously, the used startup configurations would depend on the sequence of the referenced Function Group States, if different startup configurations were used. Process **E** in Figure 7.6 is an example for such an Application Executable instance.

If different startup configurations are needed for different Function Groups, then one or more instances of the same Application Executable can be configured per Function Group.

7.5.5 State Management Architecture

7.5.5.1 State Manager

The Execution Manager provides operative mechanisms and interfaces to control the actual set of running Applications, depending on the current Machine State and Function Group States. The decision of State changes is fully given to the State Manager Application.

The State Manager is here the central point where all Applications refer to, for retrieving the current and requesting new Machine States and Function Group States.

The State Manager arbitrates Machine State and Function Group State change request, which are possibly issued by

- dedicated applications
- a vehicle state manager (possibly located on a different ECU)
- error management, e.g. the Platform Health Manager
- diagnostics

The State Manager functionality is highly system dependent and OEM specific, and AUTOSAR decided against specifying functionality like the Classic Platforms BswM for the Adaptive Platform.

However, AUTOSAR specifies some interfaces of the State Manager, so all application which might request states (see above) are portable between machines with different State Manager implementations. Also, all State Manager implementations shall use the StateClient API, so the same State Manager can be reused on different platforms.

[SWS_EM_01056] State Manager ⌈ Each unconfigured Adaptive Platform (unconfigured meaning that the platform contains the functional clusters of the Adaptive Platform Foundation and the Adaptive Platform Services, but is not yet configured as a Machine, i.e. as an instance of the Adaptive Platform) shall contain an Application, called State Manager, that provides a standardized ara::com StateManager interface, so all applications can request and retrieve Machine States and Function Group States, and that uses the StateClient::GetState and StateClient::SetState APIs to set and get Machine States and Function Group States from the Execution Manager. State requests and inquieries are routed through this default State Manager without any arbitration. It is expected, that this default State Manager will be completely replaced by an OEM specific so-

lution on configured machines, providing and using the same and possibly additional interfaces. The OEM specific State Manager can be a user-level Application. ⌋ *(RS_EM_00101)*

Therefore,

- All users of the ara::com StateManager interface are technically portable between different machines without changing the implementation.
- Using the ara::com StateManager interface over the network (e.g. by a central vehicle manager) is possible due to standardized interfaces.
- The OEM specific State Manager implementation can arbitrate all state change requests in any required way.
- If application deliveries use different names for Function Groups or Function Group States, the names can be mapped accordingly in the OEM specific State Manager before issuing state change requests to the Execution Manager, or when delivering state information back to applications. This way, the application implementation does not need to be adapted for each system environment. Manifest contents can be adapted accordingly at integration time.

An overview of the interaction of the State Manager, the Execution Manager and Applications is shown in Figure 7.7.

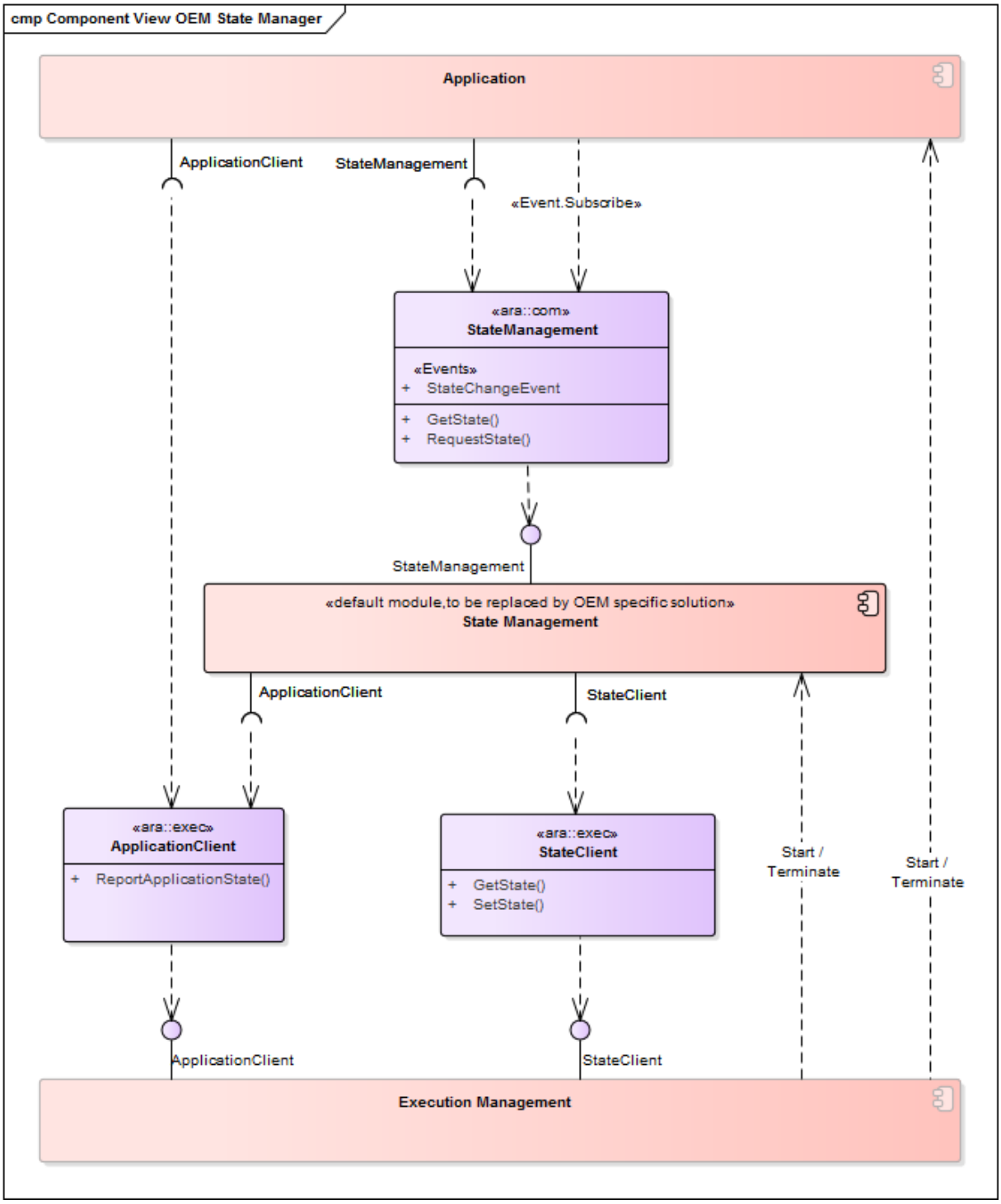

Figure 7.7: State Manager Architecture

Additional interfaces, e.g. to the `Platform Health Manager`, are not shown in this figure.

7.5.5.2 State Interaction

Figure 7.8 shows a simplified example for the interaction between different states. One can see the state transitions of a `Function Group` and the Process and Application States of one application process which references one state of this `Function Group`, ignoring possible delays and dependencies if several processes are involved. The interaction is identical if the application process references a `Machine State` instead of a `Function Group State`.

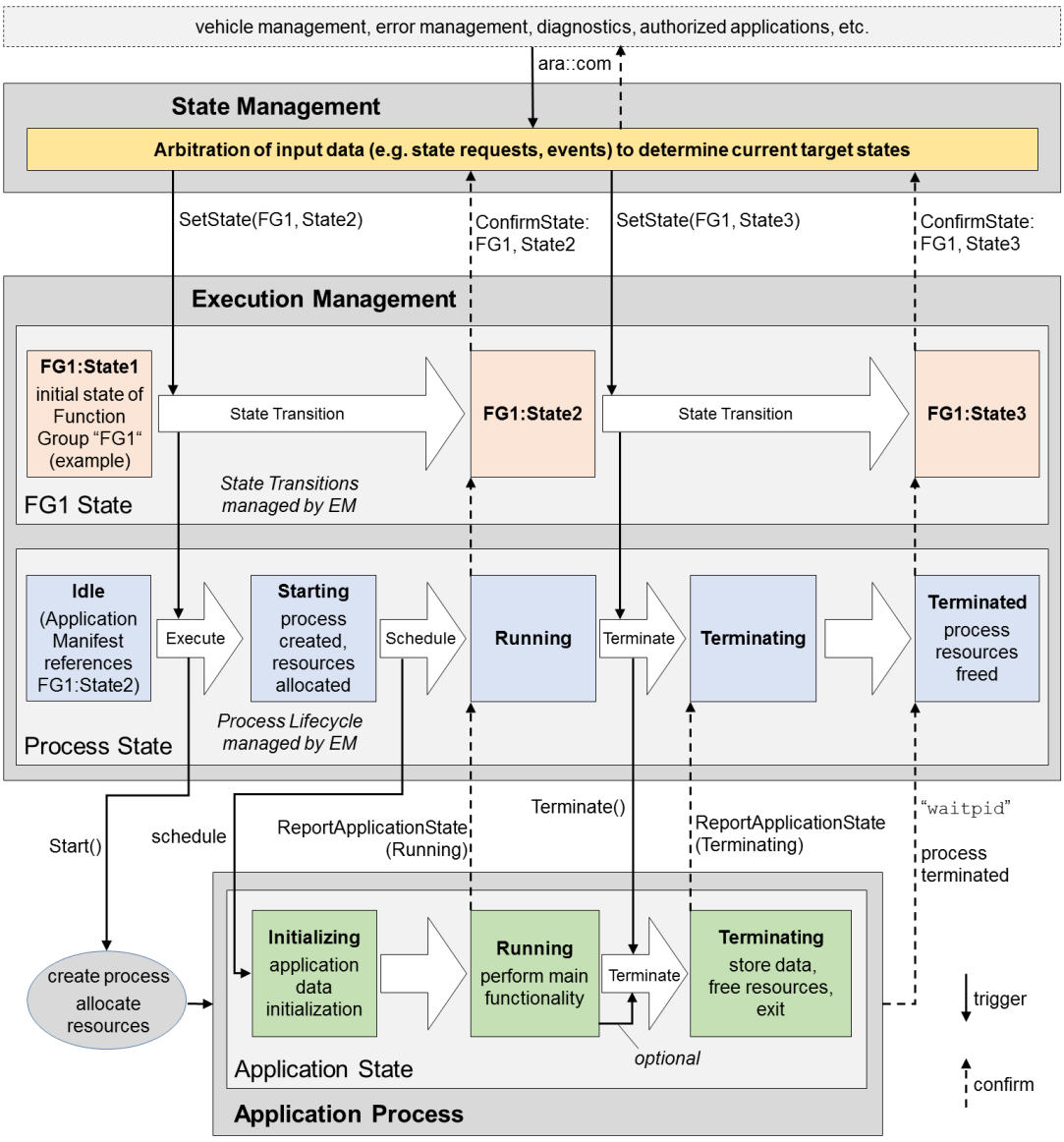

Figure 7.8: Interaction between states

7.5.6 State Change

[SWS_EM_01026] State change ⌈ The State Manager can request to change one or several Function Group States and/or the Machine State from the Execution Manager via the StateClient::SetState API. The state change request shall lead to state transitions and hereof a state change to the requested Machine State and/or Function Group States. ⌋*(RS_EM_00101)*

Machine State and Function Group State changes can be requested individually or in parallel by the State Manager. However, the following restriction applies to avoid undefined behaviour while the state transitions are performed by the Execution Manager:

[SWS_EM_01034] Deny State change request ⌈ The Execution Manager shall deny State change requests, that are received before all previously requested Machine State and/or Function Group State transitions are completed. If a request is denied, the Execution Manager shall return an error code to the requester. ⌋*(RS_EM_00101)*

[SWS_EM_01028] GetState API ⌈ The Execution Manager shall provide the interface StateClient::GetState to retrieve the current Machine State or a Function Group State. ⌋*(RS_EM_00101)*

In the following requirement, the term

"the *Process* references a *State*"

means that an instance of an Application Executable (i.e. a *Process*) has in its Application Manifest an aggregation from the Executables Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to a ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to that *State*.

A *State* can be a Machine State or a Function Group State.

CurrentStates is the collection of the Function Group States of all configured Function Groups and the Machine State at the time before one or several parallel State Transitions start.

RequestedStates is the collection of the Function Group States of all configured Function Groups and the Machine State at the time when one or several parallel State Transitions are finished.

A *SingleReferenceProcess* references in its Application Manifest either Machine States or states of one Function Group only. In Figure 7.6 this would apply to all processes except process **E**.

A *MultiReferenceProcess* references in its Application Manifest more than one type of states, e.g. Machine States and Function Group States, or states of more than one Function Group. In Figure 7.6 this would apply to process **E**. As

stated in [SWS_EM_01112], different startup configurations are not permitted in this case.

[SWS_EM_01060] State change behavior ⌈

1. For each *SingleReferenceProcess*, that

   [

   - references exactly one of the *CurrentStates*

     and

   - references none of the *RequestedStates*

     and

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

   ] or [

   - references exactly one of the *CurrentStates*

     and

   - references exactly one of the *RequestedStates*

     and

   - has different aggregated `StartupOption`s in the role `StartupConfig.startupOption`, referenced by the `ModeDependentStartupConfig`s in the role `ModeDependentStartupConfig.startupConfig`

     – with an instanceRef to the `ModeDeclaration` in the role `ModeDependentStartupConfig.machineMode` that belongs to the referenced *CurrentState*

       and

     – with an instanceRef to the `ModeDeclaration` in the role `ModeDependentStartupConfig.machineMode` that belongs to the referenced *RequestedState*.

   ],

   and

   For each *MultiReferenceProcess*, that

   [

   - references at least one of the *CurrentStates*

     and

   - references none of the *RequestedStates*

and

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

],

the Executable Process 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. The Execution Manager shall monitor the time required by the applications to terminate. In case of a timeout the following set of actions shall be performed by the Execution Management:

   - The Platform Health Manager is notified about the timeout to initiate appropriate recovery actions.
   - The timeout condition is reported back to the State Manager (as initiator of the state change) to notify that the state change cannot be fulfilled.

   The default value of the Application termination timeout shall be defined by the system integrator in the Machine Manifest. This value may be overwritten for individual application by defining the Application termination timeout parameter in the Application Manifest.

3. For each *SingleReferenceProcess*, that

   [

   - references none of the *CurrentStates*

     and

   - references exactly one of the *RequestedStates*

     and

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

   ] or [

   - references exactly one of the *CurrentStates*

     and

   - references exactly one of the *RequestedStates*

     and

   - has 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.machineMode that belongs to the referenced *CurrentState*

and

– and with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the referenced *RequestedState*.

],

and

For each *MultiReferenceProcess*, that

[

- references none of the *CurrentStates*

  and

- references at least one of the *RequestedStates*

  and

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

],

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. The Execution Manager shall monitor the time required by the Processes to reach the Running state.
5. In case the applications report the Running state within the defined timeout interval, a confirmation of the state change shall be sent to the initiator of the state change.
6. In case of a timeout the following set of actions shall be performed by the Execution Management:

   - The Platform Health Manager is notified about the timeout to initiate appropriate recovery actions.
   - The timeout condition is reported back to the State Manager (as initiator of the state change) to notify that the state change cannot be fulfilled.

   The default value of the Application startupt timeout shall be defined by the system integrator in the Machine Manifest. This value may be overwritten for individual application by defining the Application startup timeout parameter in the Application Manifest.

⌋(*RS_EM_00101*)

7.6 Application Recovery Actions

7.6.1 Overview

The Execution Management is responsible for the state dependent management of Application start/stop, so it has to have the special right to start and stop Applications. The Platform Health Management monitors Applications and could trigger a Recovery Action in case any Application behaves not within the specified parameters.

The Recovery Actions are defined by the integrator based on the software architecture requirements for the Platform Health Management and configured in the Application Manifest.

7.6.2 Recovery Actions Interfaces

[SWS_EM_01016] RestartProcess API ⌈ The Execution Management shall provide functional cluster internal interfaces to restart a specific Process on the request from the Platform Health Management. ⌋*(RS_EM_00013)*

An example for this API is RecoveryActionClient::RestartProcess.

[SWS_EM_01062] RestartProcess behaviour ⌈ The Execution Management shall restart a specific Process on the request from the Platform Health Management with the exact same startupConfig of the modeDependentStartupConfig that belongs to the to be restarted Process. ⌋*(RS_EM_00013)*

[SWS_EM_01018] `OverrideState` API ⌈ The Execution Management shall provide functional cluster internal interfaces to force the Execution Management to switch to specific Function Group States and/or to a specific Machine State on the request from the Platform Health Management. ⌋*(RS_EM_00013)*

An example for this API is RecoveryActionClient::OverrideState.

[SWS_EM_01061] OverrideState API interrupt ⌈A call of the OverrideState API shall stop any currently "ongoing" state change and process the "override" state change.⌋ *(RS_EM_00013)*

7.6.3 Integrated in the Execution Management

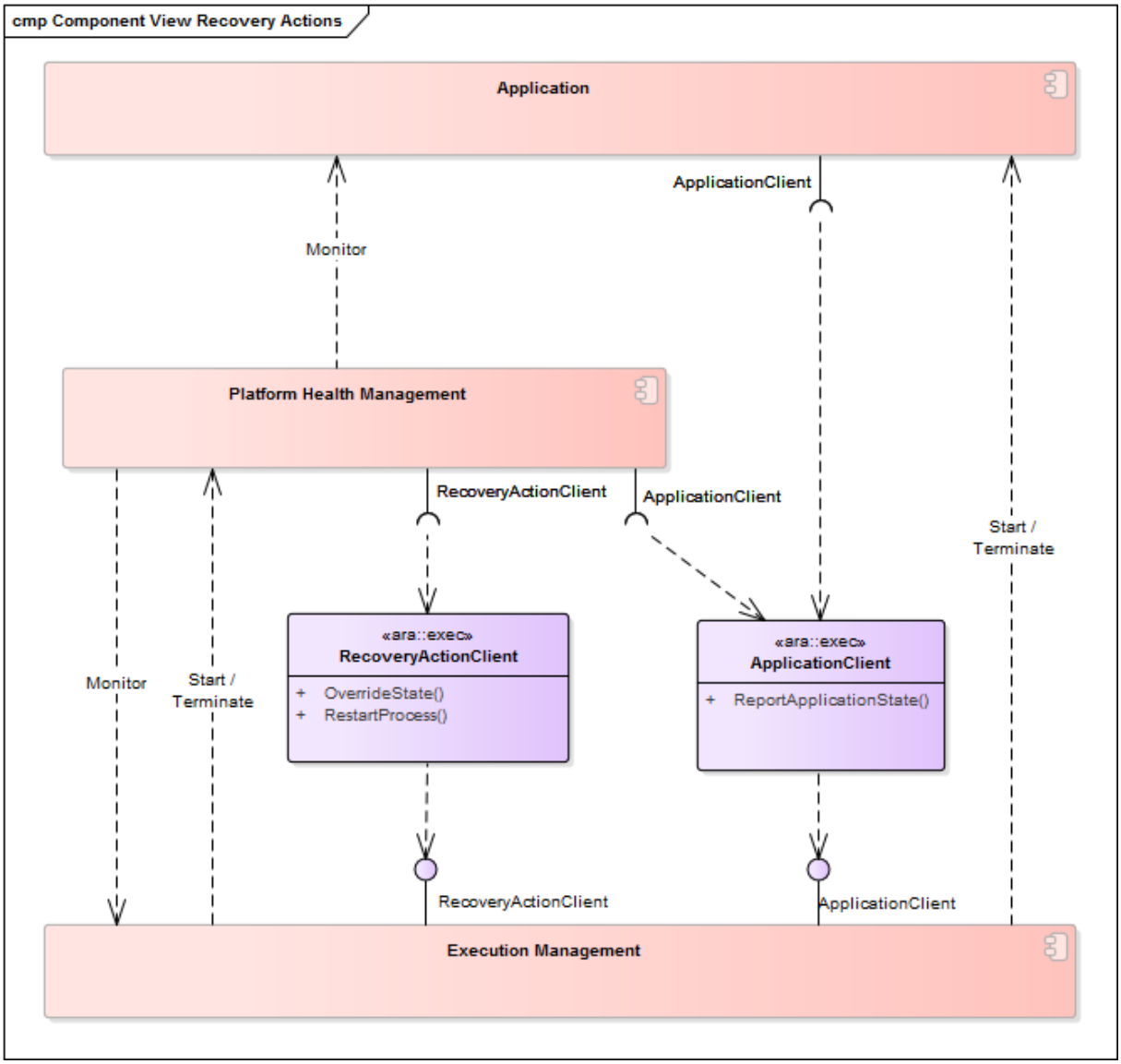

Figure 7.9: Adaptive Platform - Recovery Action Architecture

7.7 Resources and Deterministic Execution

This chapter deals with configuration and monitoring of resources and how this relates to deterministic execution.

7.7.1 Introduction

`Adaptive Platform` cannot support all mechanisms as described in this overview chapter in a standardized way, because the availability highly depends on the used Operating System. Also, some mechanisms that could be standardized will not yet be defined in this release.

7.7.1.1 Resource Configuration

In this section we give an overview on resource assignment to instances of Application Executables, i.e. processes. The main resources are:

- RAM (e.g. for code, data, thread stacks, heap)
- CPU time

Other resources like persistent storage or I/O usage are also be relevant, but will not be evaluated further in this document, which focuses on Execution Management.

The following stakeholders are involved in resource management:

- Application Developer

  The Application developer should know how much memory (RAM) and computing resources the Executable instances need to perform their tasks within a specific time. This needs to be specified in the Application description (which can be the pre-integration stage of the Application Manifest) which is handed over to the integrator. Additional constraints like a deadline for finishing a specific task, e.g. cycle time, will usually also be configured here.

  However, the exact requirements may depend on the specific use case, e.g.

  - The RAM consumption might depend on the intended use, e.g. a video filter might be configurable for different video resolutions, so the resource needs might vary within a range.
  - The computing power required depends on the processor type. i.e. the resource demands need to be converted into a computing time on that specific hardware. Possible parallel thread execution on different cores also needs to be considered here.

  In general, we need to distinguish between two resource demand values:

  - Minimum resources, which need to be guaranteed so the process can reach its Running state and perform its basic functionality.
  - Maximum resources, which might be temporarily needed and shall not be exceeded at any time, otherwise an error can be assumed.

- Integrator

The integrator knows the specific platform and its available resources and constraints, as well as other applications which may run at the same time as the processes to be configured. The integrator must assign available resources to the applications which can be active at the same time, which is closely related to State Management configuration, see section 7.5. If not enough resources are available at any given time to fulfill the maximum resource needs of all running processes, assuming they are actually used by the processes, several steps have to be considered:

- Assignment of resource criticality to processes, depending on safety and functional requirements and on the need for Deterministic Execution (see 7.7.1.3).
- Depending on the Operating System, maximum resources which cannot be exceeded by design (e.g. Linux cgroups) can be assigned to a process or a group of processes.
- A scheduling policy has to be applied, so threads of processes with high criticality get guaranteed computing time and finish before a given deadline, while threads of less critical processes might not. For details see section 7.7.2.1.
- If the summarized maximum RAM needs of all processes, which can be running in parallel at any given time, exceeds the available RAM, this cannot be solved easily by prioritization, since memory assignment to low critical processes cannot just be removed without compromising the process. However, it must be ensured that processes with high criticality have ready access to their maximum resources at any time, while lower criticality processes need to share the remaining resources. For details see 7.7.2.2.

In this release, `Adaptive Platform` provides means to configure “ResourceGroups”. One or several processes can be assigned to one of the configured ResourceGroups. Each `Adaptive Application` must be assigned to one ResourceGroup. For details see 7.7.2.

7.7.1.2 Resource Monitoring

As far as technically possible, the resources which are actually used by a process should be controlled at any given time. For the entire system, the monitoring part of this activity is fulfilled by the Operating System. For details on CPU time monitoring see 7.7.2.1. For RAM monitoring see 7.7.2.2. The monitoring capabilities depend on the used Operating System. Depending on system requirements and safety goals, an appropriate Operating System has to be chosen and configured accordingly, in combination with other monitoring mechanisms (e.g. for execution deadlines) with are provided by Platform Health Management.

Resource monitoring can serve several purposes, e.g.

- Detection of misbehaviour of the monitored process to initiate appropriate recovery actions, like process restart or state change, to maintain the provided functionality and guarantee functional safety.
- Protection of other parts of the system by isolating the erroneous processes from unaffected ones to avoid resource shortage.

For processes which are attempting to exceed their configured maximum resource needs (see 7.7.1.1), one of the following alternatives is valid:

- The resource limit violation or deadline miss is considered a failure and recovery actions may need to be initiated. Therefore the specific violation gets reported to the Platform Health Management, which then starts recovery actions which have been configured beforehand. This will be the standard option for deterministic subsystems (see 7.7.1.3).
- For processes without hard deadlines, resource violations sometimes can be mitigated without dedicated error recovery actions, e.g. by interrupting execution and continue at a later point in time.
- If the OS provides a way to limit resource consumption of a process or a group of processes by design, explicit external monitoring is usually not necessary and often not even possible. Instead, the limitation mechanisms make sure that resource availability for other parts of the system is not affected by failures within the enclosed processes. When such by-design limitation is used, monitoring mechanisms may still be used for the benefit of the platform, but are not required. Self-monitoring and out-of-process monitoring is currently out-of-scope in `Adaptive Platform`.

7.7.1.3 Deterministic Execution

In real-time systems, deterministic execution means in general, that for a given set of input data a calculation always produces consistent output within a bounded time, i.e. the behavior is reproducible.

In our case the term “calculation” can apply to execution of a thread, a process, or a group of processes. The calculation can be event-driven or cyclic, i.e. time-driven.

Determinism must be distinguished from other non-functional qualities like reliability or availability, which all deal in different ways with the statistical risk of failures. Determinism does not provide such numbers, it only defines the behavior in the absence of errors.

We can distinguish between different types of determinism:

- Time Determinism: The results of the calculation are guaranteed to be available before a given deadline.

- Data Determinism: The results only depend on the input data and are reproducible, assuming a given initial internal state.
- Full Determinism: Combination of Time and Data Determinism.

Deterministic behavior is in particular important for safety-critical systems, which may not be allowed to deviate from the specified behavior at all. Whether Time Determinism, or in addition Data Determinism is necessary to provide the required functionality depends on the system and on the safety goals.

This list exemplifies some expected use cases of the `Adaptive Platform` where such determinism is required:

- Safety goals for Highly Automated Driving (HAD) systems up to ASIL D
- High Performance Computing (HPC) demands can only be met by non automotive-grade, e.g. consumer electronics (CE), microprocessors
- Most likely no such microprocessor available for ASIL above B, at least for the parts relevant to the design
- Additional system-level safety mechanisms required to support ASIL C / D functions, e.g. software lockstep concepts. Please find more on software lockstep in 7.7.1.3.2
- Fully-deterministic processing must be supported for software lockstep concepts
- To meet HPC demands, highly predictable and reliable multi-threading must be supported
- Deterministic processing is important for integration/modeling/simulation

7.7.1.3.1 Time Determinism

Each time a calculation is started, its results are guaranteed to be available before a specified deadline. To achieve Time Determinism, sufficient and guaranteed resources (computing time, memory, service response times etc.) must be assigned to the software entities that perform the calculation.

The resources must be guaranteed at any time. Best-effort processes can request guaranteed minimum resources for base functionality, and additionally can have maximum resources specified for monitoring. If Time Determinism is requested, minimum and maximum resources are identical and must be guaranteed at any time.

If the assumptions for deterministic execution are violated, e.g. due to a deadline miss, this must be treated as an error and recovery actions must be initiated because the deterministic behavior is broken. In non deterministic “best-effort” subsystems such deadline violations or other deviations from normal behavior sometimes can be tolerated and mitigated without dedicated error management.

Fully-Deterministic behavior requires in addition Data Determinism (see below), however in many use cases Time Determinism is sufficient.

7.7.1.3.2 Data Determinism

For Data Determinism, each time a calculation is started, its results only depend on the input data. For a specific sequence of input data, the results are always exactly the same, assuming the same initial internal state. More on this in section 7.8.

A common approach to verify Data determinism in a safety context is the use of lockstep mechanisms, where execution is done simultaneously through two different paths and the result is compared to verify consistency. Hardware lockstep means that the hardware has specific equipment to make this double-/multi-execution transparent ; Software lockstep is another technique that allows providing a similar property without requiring the use of extra hardware.

Depending on the Safety Level, as well as the Safety Concept employed, software lockstep may involve executing multiple times the same software, in parallel or sequentially, but may also involve running multiple separate implementations of the same algorithm.

7.7.1.3.3 Full Determinism

For Full Determinism, each time a calculation is started, its results are available before a specified deadline and only depend on the input data, i.e. both Time and Data Determinism must be guaranteed.

In this release we restrict ourselves on supporting Full Deterministic behaviour within one process. Determinism in a cluster of processes on one or even several machines needs extensions of the Communication Management, which have not been discussed and specified yet.

Non-deterministic behavior can have many reasons, for example insufficient resources, accessing data which can be changed by other parts of the system at random times while the calculation is running, or process internal parallel execution on multiple cores, where parallel threads read or write the same data. The order in which the threads access such data will affect the result, which makes it non-deterministic (“race condition”).

A fully deterministic calculation must be designed, implemented and integrated in a way such that it is independent of processor load caused by other functions and calculation, sporadic unrelated events, race conditions, etc., i.e. it always produces the same result within a given time.

To provide full determinism for a subsystem has the following advantages:

- The deterministic subsystem shows the same behaviour on each platform which satisfies the performance and resource needs in each environment of unrelated applications. This includes development and simulation platforms.

- Due to reproducible functional behavior, many results of testing, configuration and calibration of the subsystem are valid in each environment where the subsystem is deployed on and don't need to be repeated.
- Execution of the deterministic subsystem can be replicated and results can be compared, e.g. for software lockstep concepts.

7.7.2 Resource configuration and monitoring

We need to be able to configure minimum, guaranteed resources (RAM, computing time) and maximum resources. In case Time or Full Determinism is required, the maximum resource needs are guaranteed.

7.7.2.1 CPU usage

CPU usage is represented in a process by its threads. However, several mechanisms may be available to restrict or configure the scope of execution of these threads. These mechanisms include:

- Core affinity, restricting a given thread to one or a set of cores in the system
- Scheduling policy, restricting a given thread and possibly threads inheriting the attributes, to compete in a certain way for CPU time (e.g. SCHED_RR or SCHED_FIFO)
- Scheduling group, restricting a thread to share CPU time with other threads in the same group.

While scheduling policies are not a sufficient method to guarantee Full Determinism, they contribute to improve it. Note that while the aim is to limit CPU time for a process, scheduling policies apply to threads.

Currently available POSIX-compliant Operating Systems offer the scheduling policies required by POSIX, and in most cases additional, but different and incompatible scheduling strategies. This means for now, the required scheduling properties need to be configured individually, depending on the chosen OS. Moreover, scheduling strategy is defined per thread and the POSIX standard allows for modifying the scheduling policy at runtime for a given thread, using pthread_setschedparam(). It is therefore not currently possible for the `Adaptive Platform` to enforce a particular scheduling strategy for an entire process, but only for its first thread.

The following elements can influence process and thread scheduling:

- Selection of an OS which can support the expected use cases for this platform, e.g. regarding isolation, performance, and determinism.
- Configuration of ResourceGroups in the Machine Manifest

- POSIX API calls and using C++ libraries from within the Application source code.
- Application Executable instance specific configuration at integration time in Application Manifests.
- Execution Management converts the execution requirements of the processes, as configured in the manifests, into a vendor-specific OS configuration.

In general, for deterministic behavior the required computing time is guaranteed and violations are treated as error, while best-effort subsystems are more robust and might be able to mitigate sporadic violations, e.g. by continuing the calculation at the next activation, or by providing a result of lesser quality. This means, if time (e.g. deadline or runtime budget) monitoring is in place, the reaction on deviations is different for deterministic and best-effort subsystems. In fact, it may not even be necessary to monitor best-effort subsystems, since they by definition are doing only a function that may not succeed. This leads to an architecture where monitoring is a voluntary, configured property.

7.7.2.2 Memory Budget and Monitoring

To render a function, a process requires the availability of some amount of memory for its usage (mainly code, data, heap, thread stacks). Over the course of its execution however, not all of this memory is required at all times, such that an OS can take advantage of this property to make these ranges of memory available on-demand, and provide them to other processes when the memory is no longer used. While this has clear advantages in terms of system flexibility as well as memory efficiency, it is also in the way of both Time Determinism and Full Determinism: when a range of memory that was previously unused must now be made available, the OS may have to execute some amounts of potentially-unbounded activities to make this memory available. Often, the reverse may also be happening, removing previously available (but unused) memory from the process under scope, to make it available to other processes. This is detrimental to an overall system determinism.

The Execution Management should ensure that the entire memory range that deterministic Applications may be using is available at the start and for the whole duration of the respective Application execution.

Applications not configured to be deterministic may be mapped on-demand.

In order to provide sufficient memory at the beginning of the execution of an Adaptive Application, the following properties need to be defined for each Application:

- Process heap (used for malloc())
- Memory quota (used for mapping new threads or creating kernel resources)
- Whether the Application should be fully mapped at startup

In order to further control resource usage, the following properties are also configurable for the first thread of the process:

- Scheduling policy (SCHED_RR, SCHED_FIFO)
- Stack size

Note that while the `Execution Management` will ensure the proper configuration for the first thread (that calls the main() function), it is the responsibility of the `Application` itself to properly configure secondary threads.

7.8 Deterministic Redundant Execution

As stated in 7.7.1.3, some typical future systems need high computing performance in combination with high ASIL safety goals. In this chapter we specifiy mechanisms which support deterministic multithreading to build high performance software lockstep solutions.

Formal API definitions and configuration parameters will be provided in a later release. Also, the concept of redundant execution might be extended in general in upcomming releases.

7.8.1 Redundant Execution Overview

The `Adaptive Platform` needs to provide some mechanisms to support redundant execution with sufficient isolation:

- Real-Time Execution (Time Determinism), see 7.7.1.3.1

  To ensure that execution of a given execution unit is finished at a given deadline we need:

  - appropriate scheduling policies
  - multithreading to meet high performance demand
  - deadline monitoring, execution budget monitoring and appropriate recovery action in case of violation

- Disjoint core assignment (pinning) of redundant processes to reduce common cause failures.

  [SWS_EM_01201] Core Binding ⌈ `Execution Management` shall consider assignment of one or several cores to to be used for execution of the threads of an `Executable` instance (or alternatively exclusion of one or several cores) as configured in the `Application Manifest`, see [TPS_MANI_03148]. ⌋ *(RS_EM_00008)*

- Support for recurring (e.g. cyclic) execution

  For redundant execution, execution behaviour and its budget (activation timing, computing time, computing resources) must be explicitly visible for `Execution Management`. The activation behavior can be realized by the `Execution Management` together with the `Communication Management` as required by the safety concept and specified in the manifests. `Execution Management` shall provide an API to support recurring execution (`wait_for_next_activation()`

  Details on `wait_for_next_activation()` see below.

- Data Determinism

For each execution of a given execution unit the calculated results only depend on the input data and its initial internal state, and are reproducible, i.e. the same input data always produce the same output data. In case of restart of one of a set of redundant execution units, e.g. for error recovery, the internal states (i.e. internal memory) need to be resynchronized. To do so, both redundant execution units, i.e. processes, might need to be re-initialized.

Remark: common HAD algorithms use particle filters which require random numbers! Therefore random number seeds need to be considered part of the input data.

- Support for multithreading

  For Data Determinism, parallel "worker" threads within a process should satisfy certain properties:

  – avoid race conditions

  – no exchange of data between worker threads (no communication)

  – no locks and synchronization points except common joins for all worker threads.

    Other, more complex solutions might be possible, but they increase complexity, reduce utilization, and are error-prone.

  – worker threads can be executed in parallel or sequentially in any order between common joins (unrestricted interleaving)

  – Deterministic pseudo random number generators

    For the cyclic behaviour of the worker threads, the `Execution Management` uses a deterministic and unique pseudo random number concept. "Deterministic" means, that the provided random numbers are identical for processes which are executed redundantly.

  – Deterministic timestamps

    `Execution Management` must return deterministic timestamps that represent the moment in time when the current cycle was activated and when the next cycle will be activated. Subsequent calls inside a cycle will always return the same value. The timestamps are identical for processes which are executed redundantly.

`Execution Management` shall provide an API to access a deterministic thread-pool, including "deterministic" input data (e.g. pseudo random number generator, timestamps) to be used within the process execution cycle, including the worker threads. If the number of required parallel worker threads exceeds the number of threads in the deterministic thread-pool (number to be fixed at integration time), `Execution Management` can use the threads of the pool several times sequentially, which shall be transparent to the user of the thread-pool.

The thread-pool is triggered by the main-thread of the process in a sequential order. There is no parallelism between the main-thread and the thread-pool.

A more generic thread-pool as part of a C++ library might be specified later as part of the `Operating System Interface`.

7.8.2 Redundant Execution Example

Figure 7.10 shows an example for a possible lockstep architecture. Within a process, all input data is available when execution starts.

Execution is triggered from outside the process by the `Execution Management`, depending on received data or events. Details on how to configure services that will trigger the execution will be provided in a later release.

The workload can be deployed to independent parallel threads, which are not allowed to exchange any information while they are running, i.e. they don't access data which can be altered by other threads to avoid race conditions. The threads can physically run in parallel or sequentially in any order. `Execution Management` will provide additional input like random numbers and timestamps. At the end of the execution cycle, the process waits for the next activation.

A second, redundant process gets the same input data and produces (in the absence of errors) the same result, due to full deterministic execution. An independent infrastructure, which will for now not be specified by AUTOSAR, provides the same input data for the redundant processes, triggers execution, and compares the results to detect failures (e.g. transient core errors due to radiation) in one of the redundant processes. This infrastructure layer can span over multiple hardware instances. Therefore, the infrastructure middleware will have to provide abstraction of the application deployment on the actual hardware.

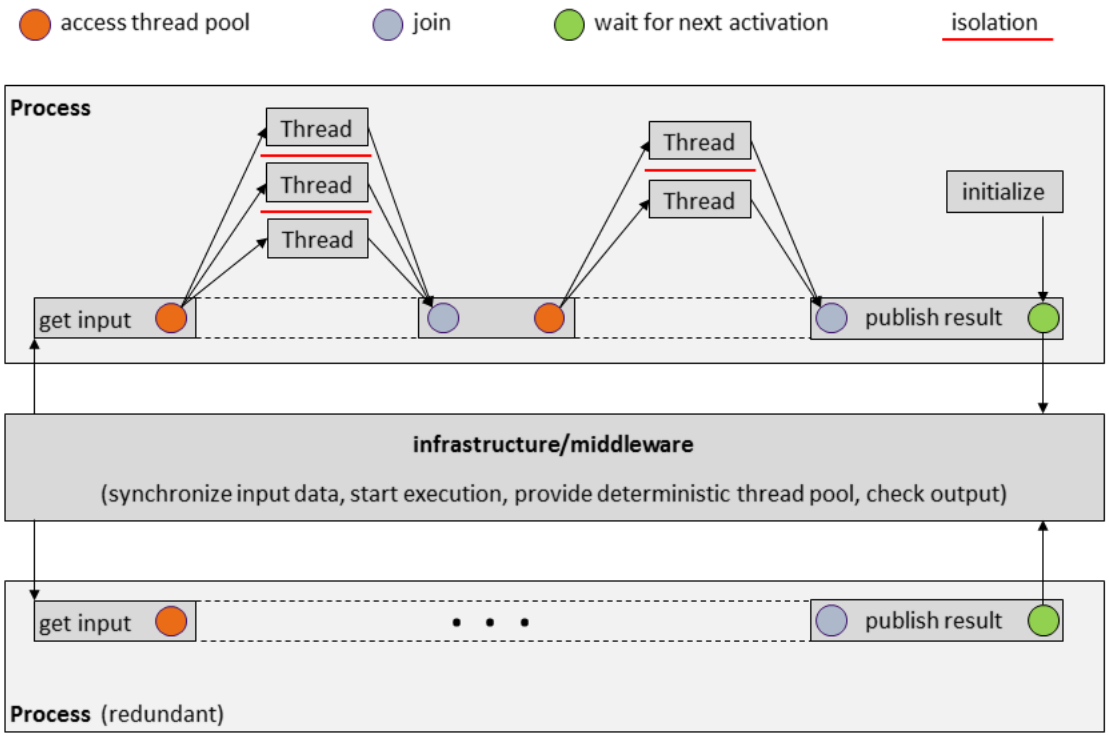

Figure 7.10: Redundant Execution

Execution Management provides an API for synchronized start of the redundant execution, and a deterministic thread-pool, including synchronized random number generation and timestamps.

7.8.3 Redundant Execution Details

Please note that content of this chapter is preliminary and subject to change.
API extension: Cyclic behaviour and worker threads:

- A thread-pool class (DeterministicSubset) provides:
  - cyclic deterministic timing
  - middleware consistence (deterministic data consistency guaranteed on cycle activation)
  - deterministic worker thread-pool with deterministic pseudo random generation
  - The thread-pool is modeled as part of the Execution Management to guarantee the deterministic behavior that includes the incorporation in the “wait_for_next_activation()”-cycle were the seeds for the pseudo random generation are provided.

  - The implementation and size of the thread-pool is hidden from the user. The Integrator decides about the size and the implementation.

```
DeterministicSubset &determinstc_subset = DeterministicSubset::
get_cyclic_behaviour_and_workerthreads(int max_thread_cnt);
```

- Cyclic deterministic timing is achieved by an Execution Management controlled main loop which is triggered from the outside via Communication Management (e.g. by external units, events generated due to the arrival of data or a timer events. Details are not in the scope of AUTOSAR).

```
enum CycleCmd { RUN , INIT, SERVICEDISCOVERY, ABORT};
```

  To be deterministic, the Application may not do service discovery on its own. The Execution Management together with Communication Management may initiate a service discovery so that in total it is deterministic

```
CycleCmd wait_for_next_activation();
```

  this blocks and returns when the next cycle is activated by the Execution Management. All input data as received via Communication Management must be guaranteed to be deterministically consistent.

- The deterministic thread-pool is used to process container elements, where each container represents a job to be computed. (e.g. based on POSIX threads.) The deterministic-distribution of the elements to individual workers is done automatically by the class-library.

```
void workers_run_on_container(Worker myrunable_obj, Container &
container);
```

  This processes all workload using the worker threads.

- Cycle timestamps

```
typedef [ARA timeformat (tbd)] ARATimeStamp;
ARATimeStamp get_current_activation_time();
```

  This returns the timestamp that represents the moment in time when the current cycle was activated. Subsequent calls inside a cycle will always return the same value.

```
ARATimeStamp get_next_activation_time();
```

  This returns the timestamp that represents the moment in time when the next cycle will be activated. Subsequent calls inside a cycle will always return the same value.

- Example worker thread runnable:

```
class MyWorker1 : public WorkerrunableBase<myContainer::
value_type, MyWorker1> {
 public:
    void worker_runable(myContainer::value_type &
container_element, WorkerThread &t) {
```

```
4                 int random_number = t.random(); // get a unique and
    deterministic pseudo-random number
5             }
6         };
```

- Worker-thread object:

```
1         class WorkerThread {
2             int random(); // returns a deterministic pseudo-random
    number that is unique for each worker
3             ...
4         };
```

- Example:

```
1   // this waits for the next trigger from EM
2   while (DeterministicSubset::CycleCmd::ABORT != (cyc_cmd =
     determinstc_subset.wait_for_next_activation())) {
3     if (DeterministicSubset::CycleCmd::INIT == cyc_cmd) {
4       // the runtime triggers an INIT to the of the APP, e.g. after
    SOTA
5       do_init();
6       do_servicediscovery();  // Discover new services
7     } else if (DeterministicSubset::CycleCmd::SERVICEDISCOVERY ==
    cyc_cmd) {
8       // the runtime is triggering service discovery, e.g. if a new
    service is available or has disappeared
9       do_servicediscovery();  // Discover new services
10    }
11
12    if (do_update()) {                              // do all updates
13      auto l_samples = l_proxy->m_myEvent.Get();  // get data
14      auto l_topic = l_skeleton.m_myEvent.Allocate();
15      // prepare delivery
16      ...
17      do_something_with_data(l_topic, l_samples, test_container);
18
19      // do something with the data and e.g. prepare container for
    parallel processing
20      MyWorker1 mw1;  // get the worker runnable
21      determinstc_subset.workers_run_on_container(mw1, test_container
    );
22
23      // process all container-elements using the workers. This will
    return when all elements are processed
24      ...
25      l_skeleton.m_myEvent.Send(l_topic);  // deliver data
26    }
27  }
```

- Worker-thread guidelines

  Worker-threads must be deterministic

  – There must not be any interaction between worker threads (e.g. no Semaphores/Mutexes, no locking/blocking)

Rationale: locking/blocking makes process runtime in-deterministic. Worker threads are meant for utilisation of runtime. If synchronization is needed an explicit join of all worker threads is necessary.

- The result of the concurrent processing with worker threads must be deterministic
  - Alternative 1:The processing of an individual worker thread must not be influenced by any other worker thread
  - Alternative 2: It must be guaranteed that, regardless of the scheduling of the threads, all results, are invariant of the timing/scheduling

    Rationale: Timing between individual threads is not guaranteed. The Operating System is scheduling threads individually.
- Individual worker threads must not access data that is influenced by other workers.

  Rationale: concurrent influencing of the same data will result in indeterminate results

7.9 Handling of Application Manifest

7.9.1 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, Function Group States 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.

7.9.2 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.

7.9.3 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 process.

7.9.4 Machine State and Function Group State

[SWS_EM_01013] Machine State and Function Group State ⌈ The Execution Management shall support the execution of specific instances of Application Executables depending on the current Machine State and Function Group States, 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 and/or Function Group 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 or Function Group 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)*

7.9.5 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 [6]. 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)*

7.9.6 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)*

7.9.7 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_00012)*

8 API specification

8.1 Type definitions

8.1.1 ApplicationState

Name:	`ApplicationState`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kRunning` `kTerminating`	`0` `1`	`--` `--`

Syntax:	`enum class ApplicationState : uint8_t {`
	`kRunning = 0,`
	`kTerminating = 1`
	`};`		

Header file:	application_client.hpp		
Description:	Defines the states of an `Application` (see 7.5.2).		

Table 8.1: ApplicationState

[SWS_EM_02000] ApplicationState Enumeration ⌈Table 8.1 describes the enumeration `ApplicationState`.⌋*(RS_EM_00103)*

8.1.2 ApplicationReturnType

Name:	`ApplicationReturnType`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kSuccess` `kGeneralError`	`0` `1`	`--` `--`

Syntax:	`enum class ApplicationReturnType : uint8_t {`
	`kSuccess = 0,`
	`kGeneralError = 1`
	`};`		

Header file:	application_client.hpp		
Description:	Defines the error codes for `ApplicationClient` operations.		

Table 8.2: ApplicationReturnType

[SWS_EM_02070] ApplicationReturnType Enumeration ⌈Table 8.2 describes the enumeration `ApplicationReturnType`.⌋*(RS_EM_00101)*

8.1.3 StateReturnType

Name:	`StateReturnType`		
Type:	`Scoped Enumeration of uint8_t`		

Range:	`kSuccess`
	`kBusy`
	`kGeneralError`
		`0`
		`1`
		`2`
			`--`
			`--`
			`--`

Syntax:	`enum class StateReturnType : uint8_t { kSuccess = 0, kBusy = 1, kGeneralError = 2 };`
Header file:	state_client.hpp
Description:	Defines the error codes for `Machine State` operations.

Table 8.3: StateReturnType

[SWS_EM_02005] StateReturnType Enumeration ⌈Table 8.3 describes the enumeration `StateReturnType`.⌋*(RS_EM_00101)*

8.1.4 RecoveryActionReturnType

Name:	`RecoveryActionReturnType`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kSuccess`	`0`	`--`
	`kGeneralError`	`1`	`--`
Syntax:	`enum class RecoveryActionReturnType : uint8_t { kSuccess = 0, kGeneralError = 1 };`		
Header file:	not specified		
Description:	Defines the error codes for `Recovery Action` operations.		

Table 8.4: RecoveryActionReturnType

Table 8.4 describes the enumeration `RecoveryActionReturnType`.

8.1.5 ResetCause

Name:	`ResetCause`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kHardReset`	`0`	This value identifies a "hard reset" condition which simulates the power-on / start-up sequence typically performed after a server has been previously disconnected from its power supply (i.e. battery).
	`kSoftReset`	`1`	This value identifies a "soft reset" condition, which causes the server to immediately restart the application program if applicable.
	`kKeyOffOnreset`	`2`	This value identifies a condition similar to the driver turning the ignition key off and back on. This reset condition should simulate a key-off-on sequence (i.e. interrupting the switched power supply).

	kRapidPowerShutdown	3	This subfunction applies to ECUs which are not ignition powered but battery powered only. Therefore a shutdown forces the sleep mode rather than a power off. Sleep means power off but still ready for wake-up (battery powered)
Syntax:	`enum class ResetCause : uint8_t { kHardReset = 0, kSoftReset = 1, kKeyOffOnreset = 2, kRapidPowerShutdown = 3 };`		
Header file:	application_client.hpp		
Description:	Defines the reset cause used by diagnostic services.		

Table 8.5: ResetCause

[SWS_EM_02041] ResetCause Enumeration ⌈Table 8.5 describes the enumeration `ResetCause`.⌋*(RS_EM_00110)*

8.2 Class definitions

8.2.1 ApplicationClient class

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

[SWS_EM_02001] ⌈ The `ApplicationClient` class shall be declared in the `application_client.hpp` header file. ⌋*(RS_EM_00103)*

[SWS_EM_02071] ⌈ An application shall not create more than a single instance of the `ApplicationClient` class. ⌋*(RS_EM_00103)*

8.2.1.1 ApplicationClient::ApplicationClient

Service name:	ApplicationClient::ApplicationClient	
Syntax:	`ApplicationClient();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	Implementation specific	In case the underlying IPC mechanism fails.
Description:	Creates an instance of ApplicationClient which opens the `Execution Management`s communication channel (e.g. POSIX FIFO) for reporting the application state. Each `Application` shall create an instance of this class to report its state.	

Table 8.6: ApplicationClient::ApplicationClient

[SWS_EM_02030] ApplicationClient::ApplicationClient API ⌈Table 8.6 describes the interface `ApplicationClient::ApplicationClient`.⌋*(RS_EM_00103)*

8.2.1.2 ApplicationClient::~ApplicationClient

Service name:	ApplicationClient::~ApplicationClient
Syntax:	`~ApplicationClient();`
Sync/Async:	Sync
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	~ApplicationClient deletes the ApplicationClient instance.

Table 8.7: ApplicationClient::~ApplicationClient

[SWS_EM_02002] ApplicationClient::~ApplicationClient API ⌈Table 8.7 describes the interface `ApplicationClient::~ApplicationClient`.⌋*(RS_EM_00103)*

8.2.1.3 ApplicationClient::ReportApplicationState

Service name:	ApplicationClient::ReportApplicationState	
Syntax:	`ApplicationReturnType ReportApplicationState( ApplicationState state );`	
Sync/Async:	Sync	
Parameters (in):	state	Value of the `Application`s state
Parameters (inout):	None	
Parameters (out):	None	
Return value:	`kSuccess`	Retrieval operation succeeded.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Interface for an `Application` to report the state to `Execution Management`.	

Table 8.8: ApplicationClient::ReportApplicationState

[SWS_EM_02003] ApplicationClient::ReportApplicationState API ⌈Table 8.8 describes the interface `ApplicationClient::ReportApplicationState`.⌋ *(RS_EM_00103)*

8.2.1.4 ApplicationClient::SetLastResetCause

Service name:	ApplicationClient::SetLastResetCause

Syntax:	`ApplicationReturnType SetLastResetCause( ResetCause cause );`	
Sync/Async:	Sync	
Parameters (in):	cause	The reset cause.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	`kSuccess`	Set operation succeeded.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Interface for an application set the last request cause.	

Table 8.9: ApplicationClient::SetLastResetCause

[SWS_EM_02042] ApplicationClient::SetLastResetCause API ⌈Table 8.9 describes the interface `ApplicationClient::SetLastResetCause`.⌋ *(RS_EM_00110)*

8.2.1.5 ApplicationClient::GetLastResetCause

Service name:	ApplicationClient::GetLastResetCause	
Syntax:	`ApplicationReturnType GetLastResetCause( ResetCause& cause );`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	cause	The reset cause.
Return value:	`kSuccess`	Retrieval operation succeeded.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Interface for an application retrieve the last request cause.	

Table 8.10: ApplicationClient::GetLastResetCause

[SWS_EM_02043] ApplicationClient::GetLastResetCause API ⌈Table 8.10 describes the interface `ApplicationClient::GetLastResetCause`.⌋ *(RS_EM_00110)*

8.2.2 StateClient class

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

The `Application` responsible for managing and controlling the states, `State Management` has to instantiate this class.

Note on Platform internal APIs:
When State Management becomes an own functional cluster instead of a user-level application, the APIs `StateClient::GetState` and `StateClient::SetState` are platform internal APIs not exposed to user-level applications. Therefore they are shown as examples how an implementation could look like. It shall then not be necessary that a functional cluster implementation has to implement it exactly as defined in this specification.

[SWS_EM_02006] ⌈ The `StateClient` class shall be declared in the `state_client.hpp` header file. ⌋*(RS_EM_00101)*

8.2.2.1 StateClient::StateClient

Service name:	StateClient::StateClient	
Syntax:	`StateClient();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	Implementation specific	In case the underlying IPC mechanism fails.
Description:	Creates an instance of StateClient which opens the `Execution Management`s communication channel (e.g. POSIX FIFO) for retrieving or requesting `Machine State`s and/or `Function Group State`s from/-to `Execution Management`.	

Table 8.11: StateClient::StateClient

[SWS_EM_02007] StateClient::StateClient API ⌈Table 8.11 describes the interface `StateClient::StateClient`.⌋*(RS_EM_00101)*

8.2.2.2 StateClient::~StateClient

Service name:	StateClient::~StateClient
Syntax:	`~StateClient();`
Sync/Async:	Sync
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	~StateClient deletes the StateClient instance.

Table 8.12: StateClient::~StateClient

[SWS_EM_02008] StateClient::~StateClient API ⌈Table 8.12 describes the interface `StateClient::~StateClient`.⌋*(RS_EM_00101)*

8.2.2.3 StateClient::GetState

Service name:	StateClient::GetState	
Syntax:	`StateReturnType GetState( std::string functionGroup, std::string &state );`	
Sync/Async:	Sync	
Parameters (in):	functionGroup	Name of the `Function Group` or the string "MachineState" to retrieve the current `Machine State`.
Parameters (inout):	None	
Parameters (out):	state	String containing the current `Function Group State` of the given `Function Group` or the current `Machine State` of the given string "MachineState" for the `functionGroup` argument. Empty string if return value is different to `kSuccess`.
Return value:	`kSuccess`	Retrieval operation succeeded.
	`kBusy`	`Execution Management` is busy and cannot respond.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Retrieve the current State of a `Function Group` or `Machine State` from the `Execution Management`.	

Table 8.13: StateClient::GetState

[SWS_EM_02047] StateClient::GetState API ⌈Table 8.13 describes the interface `StateClient::GetState`.⌋*(RS_EM_00101)*

[SWS_EM_02072] Retrieving Machine State ⌈ When the value of the `functionGroup` argument is "MachineState", the API `StateClient::GetState` shall return the current `Machine State` within the `state` argument.⌋*(RS_EM_00101)*

[SWS_EM_02044] Machine State change in progress ⌈ When `Execution Management` performs a `Machine State` change and the requested value of the parameter `functionGroup` argument is "MachineState", the API `StateClient::GetState` shall return `kBusy`. ⌋*(RS_EM_00101)*

[SWS_EM_02073] Retrieving Function Group State ⌈ When the value of the `functionGroup` argument is different from "MachineState", the API `StateClient::GetState` shall return the current state of the given `Function Group` within the `state` argument.⌋*(RS_EM_00101)*

[SWS_EM_02048] Function Group State change in progress ⌈ When `Execution Management` performs a `Function Group State` change that is being requested

with the same Function Group as requested with parameter functionGroup, the API StateClient::GetState shall return kBusy. ⌋(*RS_EM_00101*)

[SWS_EM_02049] State change failed ⌈ When a Function Group State change or Machine State change failed in the Execution Management, the API StateClient::GetState shall return kGeneralError. ⌋(*RS_EM_00101*)

[SWS_EM_02050] State change successful ⌈ When the Execution Management successfully processed the request and is able to return the current Function Group State or Machine State, the API StateClient::GetState shall return kSuccess. ⌋(*RS_EM_00101*)

8.2.2.4 StateClient::SetState

Service name:	StateClient::SetState	
Syntax:	`StateReturnType SetState( std::string functionGroup, std::string state );`	
Sync/Async:	Sync	
Parameters (in):	functionGroup	Requested Function Group or the string "MachineState" to request a Machine State.
	state	State of the Function Group or Machine State to set.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	kSuccess	Set operation succeeded.
	kBusy	Execution Management is busy and cannot respond.
	kGeneralError	GeneralError
Exceptions:	None	
Description:	Requests a new Function Group State or Machine State at the Execution Management.	

Table 8.14: StateClient::SetState

[SWS_EM_02054] StateClient::SetState API ⌈Table 8.14 describes the interface StateClient::SetState.⌋(*RS_EM_00101*)

[SWS_EM_02074] Setting Machine State ⌈ When the value of the functionGroup argument is "MachineState", the API StateClient::SetState shall set the Machine State to the value given by the state argument.⌋(*RS_EM_00101*)

[SWS_EM_02051] Machine State change in progress ⌈ When Execution Management performs a Machine State change and the requested value of the parameter functionGroup argument is "MachineState", the API StateClient::SetState shall return kBusy. ⌋(*RS_EM_00101*)

[SWS_EM_02075] Setting Function Group State ⌈ When the value of the functionGroup argument is different from "MachineState", the API StateClien-

t::SetState shall set the state of the given Function Group to the value given by the state argument.⌋(*RS_EM_00101*)

[SWS_EM_02055] Function Group State change in progress ⌈ When Execution Management performs Function Group State change that is being requested with the same Function Group as requested with parameter functionGroup, the API StateClient::SetState shall return kBusy. ⌋(*RS_EM_00101*)

[SWS_EM_02056] State change failed ⌈ The API StateClient::SetState shall return kGeneralError when other or unspecified errors occur. ⌋(*RS_EM_00101*)

[SWS_EM_02057] State change successful ⌈ When Execution Management succeeds with the Function Group State or Machine State change request, the API StateClient::SetState shall return kSuccess. ⌋(*RS_EM_00101*)

8.2.3 RecoveryActionClient class

The `RecoveryActionClient` class provides the functionality for the `Platform Health Management` to restart a `Process` or to force a switch to a specific `Machine State` / `Function Group State`.

The `Platform Health Management` responsible for managing and controlling the `Recovery Action`s, has to instantiate this class.

Note on Platform internal APIs: The APIs `RecoveryActionClient::RestartProcess` and `RecoveryActionClient::OverrideState` are platform internal APIs not exposed to user-level applications. Therefore they are shown as examples how an implementation could look like. It shall not be necessary that a functional cluster implementation has to implement it exactly as defined in this specification.

8.2.3.1 RecoveryActionClient::RecoveryActionClient

Service name:	RecoveryActionClient::RecoveryActionClient	
Syntax:	`RecoveryActionClient();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	Implementation specific	In case the underlying IPC mechanism fails.
Description:	Creates an instance of RecoveryActionClient which opens the `Execution Management`s communication channel (e.g. POSIX FIFO) to a restart a `Process` or to force a switch to a specific `Machine State` / `Function Group State`.	

Table 8.15: RecoveryActionClient::RecoveryActionClient

8.2.3.2 RecoveryActionClient::~RecoveryActionClient

Service name:	RecoveryActionClient::~RecoveryActionClient
Syntax:	`~RecoveryActionClient();`
Sync/Async:	Sync
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	~RecoveryActionClient deletes the RecoveryActionClient instance.

Table 8.16: RecoveryActionClient::~RecoveryActionClient

8.2.3.3 RecoveryActionClient::RestartProcess

Service name:	RecoveryActionClient::RestartProcess	
Syntax:	`RecoveryActionReturnType RestartProcess( pid_t processId );`	
Sync/Async:	Sync	
Parameters (in):	processId	ID of the `Process` that shall be restarted.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	`kSuccess`	Retrieval operation succeeded.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Interface for the `Platform Health Management` to request a `Process` restart.	

Table 8.17: RecoveryActionClient::RestartProcess

8.2.3.4 RecoveryActionClient::OverrideState

Service name:	RecoveryActionClient::OverrideState	
Syntax:	`RecoveryActionReturnType OverrideState( std::string functionGroup, std::string state );`	
Sync/Async:	Sync	
Parameters (in):	functionGroup	Requested `Function Group`.
	state	State of the `Function Group` to set.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	`kSuccess`	Retrieval operation succeeded.
	`kGeneralError`	GeneralError
Exceptions:	None	
Description:	Interface for the `Platform Health Management` to override a `Machine State` or a `Function Group State`. Please note that a termination request that may be required to be send to an application, should be delayed until this application report Running State.	

Table 8.18: RecoveryActionClient::OverrideState

9 Service Interfaces

This chapter lists all provided and required service interfaces of the Execution Management.

9.1 Service Type definitions

9.1.1 State_StatusType

Name	State_StatusType		
Kind	Struct		
Description	This data structure contains the Function Group State or Machine State information.		
Members	**Name**	**Type**	**Description**
	FunctionGroup	std::string	Name of the Function Group or the string "MachineState" in case of a Machine State.
	State	std::string	String containing the current Function Group State of the given Function Group or the current Machine State.

9.2 State Management Interface

9.2.1 Methods

Name	RequestState		
Description	Requests a new Function Group State or Machine State.		
Parameters	FunctionGroup	**Description**	Requested Function Group or the string "MachineState" to request a Machine State.
		Type	std::string
		Direction	IN

	State	***Description***	New requested state of the Function Group or Machine State.
		Type	std::string
		Direction	IN

Name	GetState		
Description	Retrieves the current state of a Function Group or Machine State.		
Parameters	FunctionGroup	***Description***	Name of the Function Group or the string "MachineState" to retrieve the current Machine State.
		Type	std::string
		Direction	IN
	State	***Description***	String containing the current Function Group State of the given Function Group or the current Machine State.
		Type	std::string
		Direction	OUT

9.2.2 Events

This service interface provides a notification event triggered by a state change.

Name	StateChangeEvent
Description	Notification about Function Group State or Machine State changes. This event is triggered whenever a Function Group State or Machine State change happens.
Type	State_StatusType

A Not applicable requirements

[SWS_EM_NA] ⌈ These requirements are not applicable as they are not within the scope of the 2017-10 release. ⌋*(RS_EM_00003, RS_EM_00004, RS_EM_00005, RS_EM_00006, RS_EM_00007, RS_EM_00050, RS_EM_00051, RS_EM_00052, RS_EM_00053)*

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::Deployment::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. Example: --version=1.0 --help **Tags:** atp.EnumerationValue=1
commandLineShortForm	Short form of command line option. Example: -v 1.0 -h **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::Deployment::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
functionGroup	ModeDeclaration	*	iref	This represent the applicable functionGroup. **Tags:** atp.Status=draft
machineMode	ModeDeclaration	*	iref	This represent the applicable machineMode. **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::Deployment::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::Deployment::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	1	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 for the initial thread of the application.
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::Deployment::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.

optionNam e	String	0..1	attr	This attribute defines option name.

Table B.6: StartupOption