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	19-03

Document Change History

Date	Release	Changed by	Description
2019-03-29	19-03	AUTOSAR Release Management	• Refinement of State Management semantics • Document structure modified to reflect current template
2018-10-31	18-10	AUTOSAR Release Management	• Refinement of Deterministic Execution • Updated Process lifecycle to clarify Process and Execution States • Updated Application Recovery Actions
2018-03-29	18-03	AUTOSAR Release Management	• Deterministic Execution • Resource Limitation • State Management • Fault Tolerance elaboration
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

Known Limitations

This chapter lists known limitations of Execution Management and their relation to this release of the AUTOSAR Adaptive Platform with the intent to provide an indication how Execution Management within the context of the AUTOSAR Adaptive Platform will evolve in future releases.

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

Section 7.7 Resource Limitation and Section 7.8 Fault Tolerance – these sections have been expanded in this release but are not complete. In particular the contents will be expanded with more properties and formal requirements in the next release.

The following functionality is not specified in this release:

- Support of a Trusted Platform ([RS_EM_00014]).

Section 6.1 details requirements from Execution Management Requirement Specification [1] 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 AUTOSAR Adaptive Platform.

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

1 Introduction and functional overview

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

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

Chapter 7 describes how Execution Management concepts are realized within the AUTOSAR Adaptive Platform.

Chapter 8 documents the Execution Management Application Programming Interface (*API*). Inter-functional cluster (IFC) interfaces are described in Appendix B.

1.1 What is Execution Management?

Execution Management is the functional cluster within the Adaptive Platform Foundation that is responsible for platform initialization and the startup and shutdown of Adaptive Applications. It performs these tasks using information contained within one or more Manifest files such as when and how Executables should be started.

The Execution Management functional cluster is part of the AUTOSAR Adaptive Platform. However, the AUTOSAR Adaptive Platform is usually not exclusively used within a single AUTOSAR System as the vehicle is also equipped with a number of ECUs developed on the *AUTOSAR Classic Platform*. The System design for the entire vehicle will therefore cover both *AUTOSAR Classic Platform* ECUs as well as AUTOSAR Adaptive Platform Machines.

1.2 Interaction with AUTOSAR Runtime for Adaptive

The set of programming interfaces to the Adaptive Applications is called AUTOSAR Runtime for Adaptive (ARA). The interfaces that constitute ARA include those of Execution Management specified in Chapter 8. Note that interfaces which are exclusively accessed within the AUTOSAR Adaptive Platform are defined as inter-functional cluster (IFC) interfaces (see Appendix B), which are not part of 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 user-level Applications. The launching order is derived by Execution Management according to the specification defined in this document to ensure proper startup of the AUTOSAR Adaptive Platform.

The Adaptive AUTOSAR Services are provided via mechanisms provided by the Communication Management functional cluster [2] of the Adaptive Platform Foundation. In order to use the Adaptive AUTOSAR Services, the functional clusters in the Adaptive Platform Foundation must be properly initialized beforehand. Please refer to the respective specifications regarding more information on Communication Management.

2 Acronyms and abbreviations

All technical terms used throughout this document – except the ones listed here – can be found in the official [3] AUTOSAR Glossary or [4] TPS Manifest Specification.

Certain requirements necessitate the specification of mandatory whitespace. This is indicated by '␣' in the requirement text.

Term	Description
Process	A Process is a loaded instance of an Executable to be executed on a Machine. This document also uses the term process without emphasis to refer to the POSIX concept of a running process.
Self-terminating Process	A type of Process that self initiate termination procedure, please note that for a standard Process this procedure is initiated by Execution Management.
Execution Dependency	Dependencies between Executable instances can be configured to define a sequence for starting and terminating them.
Execution Management	The element of the AUTOSAR Adaptive Platform responsible for the ordered startup and shutdown of the AUTOSAR Adaptive Platform and the Applications.
State Management	The element of AUTOSAR Adaptive Platform defining modes of operation. It allows flexible definition of functions which are active on the platform at any given time.
Function Group	A Function Group is a set of coherent Processes, which need to be controlled consistently. Depending on the state of the Function Group, Processes are started or terminated. Processes can belong to more than one Function Group. "MachineState" is a Function Group with a predefined name, which is mainly used to control Machine lifecycle and Processes of platform level Applications. Other Function Groups are sort of general purpose tools used (for example) to control Processes of user level Applications.
Function Group State	The state of a Function Group (except "MachineState"). It defines a set of active Applications for any certain situation. The set of Function Groups and their states is machine specific and is deployed as part of the Machine Manifest.
Machine State	The state of Function Group "MachineState" with some predefined states (Startup/Shutdown/Restart).
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.
Communication Management	A Functional Cluster within the Adaptive Platform Foundation
Execution Manifest	Manifest file to configure execution of an Adaptive Application.
Machine Manifest	Manifest file to configure a Machine.
Operating System	Software responsible for managing Processes on a Machine and for providing an interface to hardware resources.
ResourceGroup	Configuration element to enable restrictions on resources uses by Adaptive Applications running in the group.

ExecutionClient	Adaptive Application interface to Execution Management.
DeterministicClient	Adaptive Application interface to Execution Management to support control of the process-internal cycle, a deterministic worker pool, activation time stamps and random numbers.
Platform Health Management	A Functional Cluster within the Adaptive Platform Foundation
Recovery Action	Actions defined by the integrator to control Adaptive Application error recovery.
Process State	Lifecycle state of a Process
Service Instance Manifest	Manifest file to configure Service usage of an Adaptive Application.
Trusted Platform	An execution platform supporting a continuous chain of trust from boot through to application supporting authentication (that all code executed is from the claimed source) and integrity validation (that prevents tampered code/data from being executed).

Table 2.1: Technical Terms

The following technical terms used throughout this document are defined in the official [3] AUTOSAR Glossary or [4] TPS Manifest Specification – they are repeated here for tracing purposes.

Term	Description
Adaptive Application	see [3] AUTOSAR Glossary
Application	see [3] AUTOSAR Glossary
AUTOSAR Adaptive Platform	see [3] AUTOSAR Glossary
Adaptive Platform Foundation	see [3] AUTOSAR Glossary
Adaptive Platform Services	see [3] AUTOSAR Glossary
Manifest	see [3] AUTOSAR Glossary
Executable	see [3] AUTOSAR Glossary
Functional Cluster	see [3] AUTOSAR Glossary
Machine	see [3] AUTOSAR Glossary
Service	see [3] AUTOSAR Glossary
Service Interface	see [3] AUTOSAR Glossary
Service Discovery	see [3] AUTOSAR Glossary

Table 2.2: Glossary-defined 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] Requirements on Execution Management
AUTOSAR_RS_ExecutionManagement

[2] Specification of Communication Management
AUTOSAR_SWS_CommunicationManagement

[3] Glossary
AUTOSAR_TR_Glossary

[4] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[5] General Specification of Adaptive Platform
AUTOSAR_SWS_General

[6] Specification of Operating System Interface
AUTOSAR_SWS_OperatingSystemInterface

[7] Specification of Persistency
AUTOSAR_SWS_Persistency

[8] Specification of Platform Health Management for Adaptive Platform
AUTOSAR_SWS_PlatformHealthManagement

[9] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology

[10] Specification of State Management
AUTOSAR_SWS_StateManagement

[11] Guidelines for using Adaptive Platform interfaces
AUTOSAR_EXP_AdaptivePlatformInterfacesGuidelines

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

[13] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, 'Basic Concepts and Taxonomy of Dependable and Secure Computing', IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004

3.2 Related Standards and Norms

AUTOSAR provides a general specification [5] which is also applicable for Execution Management. The specification SWS General shall be considered as additional and required specification for implementation of Execution Management.

4 Constraints and assumptions

4.1 Applicability to Car Domains

No restrictions to applicability.

5 Dependencies to other functional clusters

5.1 Platform Dependencies

5.1.1 Operating System Interface

Execution Management is dependent on the Operating System Interface [6]. The OSI is used to control specific aspects of Application execution, for example, to set scheduling parameters or to execute an Application.

5.1.2 Persistency

Execution Management is dependent on the Persistency [7] functional cluster. Persistency is used to access persistent storage.

5.1.3 Dependencies on Platform Health Management

Execution Management is dependent on the Platform Health Management [8] functional cluster. The Platform Health Management Interfaces are used by Execution Management for notifying a state change of a Process.

5.2 Other Dependencies

Execution Management may dependent on the Operating System beyond the Operating System Interface [6], e.g to control the core affinity of processes (refer 7.7.3.2).

6 Requirements tracing

The following tables reference the requirements specified in [1] 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_AP_00115]	Namespaces.	[SWS_EM_NA]
[RS_AP_00124]	Variable names.	[SWS_EM_NA]
[RS_AP_00130]	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment.	[SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249]
[RS_AP_00131]	Use of verbal forms to express requirement levels.	[SWS_EM_01000] [SWS_EM_01001] [SWS_EM_01002] [SWS_EM_01003] [SWS_EM_01004] [SWS_EM_01005] [SWS_EM_01006] [SWS_EM_01012] [SWS_EM_01013] [SWS_EM_01014] [SWS_EM_01015] [SWS_EM_01016] [SWS_EM_01018] [SWS_EM_01023] [SWS_EM_01024] [SWS_EM_01025] [SWS_EM_01026] [SWS_EM_01028] [SWS_EM_01030] [SWS_EM_01032] [SWS_EM_01033] [SWS_EM_01034] [SWS_EM_01041] [SWS_EM_01042] [SWS_EM_01043] [SWS_EM_01050] [SWS_EM_01051] [SWS_EM_01053] [SWS_EM_01055] [SWS_EM_01060] [SWS_EM_01061] [SWS_EM_01062] [SWS_EM_01063] [SWS_EM_01064] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01068] [SWS_EM_01070] [SWS_EM_01071] [SWS_EM_01072] [SWS_EM_01073] [SWS_EM_01074] [SWS_EM_01075] [SWS_EM_01076] [SWS_EM_01077] [SWS_EM_01107] [SWS_EM_01109] [SWS_EM_01110] [SWS_EM_01301] [SWS_EM_01302] [SWS_EM_01303] [SWS_EM_01304] [SWS_EM_01305] [SWS_EM_01306] [SWS_EM_01308] [SWS_EM_01310] [SWS_EM_01311] [SWS_EM_01312] [SWS_EM_01313] [SWS_EM_01351] [SWS_EM_01352] [SWS_EM_01353] [SWS_EM_01400] [SWS_EM_02044] [SWS_EM_02049] [SWS_EM_02050] [SWS_EM_02056] [SWS_EM_02057] [SWS_EM_02058] [SWS_EM_02076] [SWS_EM_02077]

Requirement	Description	Satisfied by
		[SWS_EM_02102] [SWS_EM_02103] [SWS_EM_02104] [SWS_EM_02106] [SWS_EM_02107] [SWS_EM_02108] [SWS_EM_02109] [SWS_EM_02241] [SWS_EM_02242] [SWS_EM_02243] [SWS_EM_02244] [SWS_EM_02245] [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249] [SWS_EM_02250] [SWS_EM_02251] [SWS_EM_02252] [SWS_EM_02253] [SWS_EM_02254] [SWS_EM_02255] [SWS_EM_02256]
[RS_AP_00132]	Usage of noexcept keyword.	[SWS_EM_NA]
[RS_EM_00002]	Execution Management shall set-up one process for the execution of each Process .	[SWS_EM_01014] [SWS_EM_01015] [SWS_EM_01041] [SWS_EM_01042] [SWS_EM_01043]
[RS_EM_00005]	Execution Management shall support the configuration of OS resource budgets for Process and groups of Processes .	[SWS_EM_02102] [SWS_EM_02103] [SWS_EM_02106] [SWS_EM_02107] [SWS_EM_02108] [SWS_EM_02109]
[RS_EM_00008]	Execution Management shall support the binding of all threads of a given Process to a specified set of processor cores.	[SWS_EM_02104]
[RS_EM_00009]	Only Execution Management shall start Processes .	[SWS_EM_01030] [SWS_EM_01033]
[RS_EM_00010]	Execution Management shall support multiple instances of Executables .	[SWS_EM_01012] [SWS_EM_01072] [SWS_EM_01073] [SWS_EM_01074] [SWS_EM_01075] [SWS_EM_01076] [SWS_EM_01077] [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249]
[RS_EM_00011]	Execution Management shall support self-initiated graceful shutdown of Processes .	[SWS_EM_01005]
[RS_EM_00013]	Execution Management shall support configurable recovery actions.	[SWS_EM_01016] [SWS_EM_01018] [SWS_EM_01061] [SWS_EM_01062] [SWS_EM_01063] [SWS_EM_01064] [SWS_EM_02076] [SWS_EM_02077]
[RS_EM_00014]	Execution Management shall support a Trusted Platform.	[SWS_EM_NA]
[RS_EM_00050]	Execution Management shall perform Machine -wide coordination of Processes .	[SWS_EM_NA]
[RS_EM_00051]	Execution Management shall provide APIs to the Process for configuring external trigger conditions for its activities.	[SWS_EM_NA]
[RS_EM_00052]	Execution Management shall provide APIs to the Process for configuring cyclic triggering of its activities.	[SWS_EM_01301] [SWS_EM_01302] [SWS_EM_01303] [SWS_EM_01304] [SWS_EM_01351] [SWS_EM_01352] [SWS_EM_01353] [SWS_EM_02201] [SWS_EM_02210] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02216]

Requirement	Description	Satisfied by
[RS_EM_00053]	Execution Management shall provide APIs to the Process to support deterministic redundant execution of Processes .	[SWS_EM_01305] [SWS_EM_01306] [SWS_EM_01308] [SWS_EM_01310] [SWS_EM_01311] [SWS_EM_01312] [SWS_EM_01313] [SWS_EM_02202] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02220] [SWS_EM_02225] [SWS_EM_02230] [SWS_EM_02235]
[RS_EM_00100]	Execution Management shall support the ordered startup and shutdown of Processes .	[SWS_EM_01000] [SWS_EM_01001] [SWS_EM_01050] [SWS_EM_01051] [SWS_EM_01400]
[RS_EM_00101]	Execution Management shall support 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_01033] [SWS_EM_01034] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01068] [SWS_EM_01107] [SWS_EM_01109] [SWS_EM_01110] [SWS_EM_02044] [SWS_EM_02049] [SWS_EM_02050] [SWS_EM_02056] [SWS_EM_02057] [SWS_EM_02058] [SWS_EM_02070] [SWS_EM_02241] [SWS_EM_02242] [SWS_EM_02245] [SWS_EM_02250] [SWS_EM_02251] [SWS_EM_02252] [SWS_EM_02253] [SWS_EM_02254] [SWS_EM_02255] [SWS_EM_02256]
[RS_EM_00103]	Execution Management shall support Process 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_01070] [SWS_EM_01071] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02030] [SWS_EM_02243] [SWS_EM_02244]
[RS_EM_00111]	Execution Management shall assist identification of Processes during Machine runtime.	[SWS_EM_NA]
[RS_EM_NA]		[SWS_EM_NA]

6.1 Not applicable requirements

[SWS_EM_NA]{DRAFT} ⌈ These requirements are not applicable as they are not within the scope of this release. ⌋*(RS_EM_00014, RS_EM_00050, RS_EM_00051, RS_AP_00115, RS_AP_00124, RS_AP_00132, RS_EM_00111, RS_EM_NA)*

7 Functional specification

Execution Management is a functional cluster contained in the Adaptive Platform Foundation. 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. In particular, Execution Management is responsible for configuring the Operating System to perform run-time scheduling and resource monitoring of Applications.

This chapter describes the functional behavior of Execution Management.

- Section 7.1 presents an introduction to key terms within Execution Management focusing on the relationship between Application, Executable, and Process.
- Section 7.2 covers the core Execution Management run-time responsibilities including the start of Applications.
- Section 7.3 describes the lifecycle of Applications including Process state transitions and startup / shutdown sequences.
- Section 7.4 covers several topics related to State Management within Execution Management including execution, Machine and Function Group state management.
- Section 7.5 describes how Application error recovery actions are specified during integration.
- Section 7.6 documents support provided by Execution Management *Deterministic* execution such that given the same input and internal state, a calculation will always produce the same output.
- Section 7.7 describes how Execution Management supports resource management including the limitation of usage of CPU and memory by an Application.
- Section 7.8 provides an introduction to Fault Tolerance strategies in general. This section will be expanded in a future release to describe how such strategies are realized within Execution Management.

7.1 Technical Overview

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

7.1.1 Terms

Before discussing the concepts of Application, Executable, and Process it is useful to present an overview of the terms so that the more detailed dicussions have the required context.

Application – An implementation that resolves a set of coherent functional requirements and is the result of functional development. An Application is the unit of delivery for Machine specific configuration and integration.

Executable – Part of an Application. It consists of executable code (with exactly one entry point) created at integation time that can be deployed and installed on a Machine. An Application may consist of one or more Executables, each of which can be deployed to different Machines.

Process – A Process represents a started instance of an Executable. At run-time Execution Management implements a Process using an OS process.

Execution Manifest – An Execution Manifest is created at integration time and deployed onto a Machine together with the Executable to which it is attached. It supports the integration of the Executable code and describes the configuration properties (startup parameters, resource group assignment etc.) of each Process, i.e. started instance of that Executable.

Machine Manifest – The Machine Manifest holds all configuration information which cannot be assigned to a specific Executable or Process.

7.1.2 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 and execution (e.g. a formal model description based on the AUTOSAR meta model, test cases, etc.).

Applications can be located on user level above the middleware or can implement functional clusters of the AUTOSAR Adaptive Platform (located on the level of the middleware), see [TPS_MANI_01009] in [4].

In general, an Application, whether user-level or platform-level, is treated the same by Execution Management and can use all mechanisms and APIs provided by the Operating System and other functional clusters of the AUTOSAR Adaptive Platform. However in doing so it potentially restricts its portability to other implementations of the AUTOSAR Adaptive Platforms.

7.1.3 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 the use of APIs standardized by AUTOSAR and needs to follow specific coding guidelines to allow reallocation between different implementations of the AUTOSAR Adaptive Platform.

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.

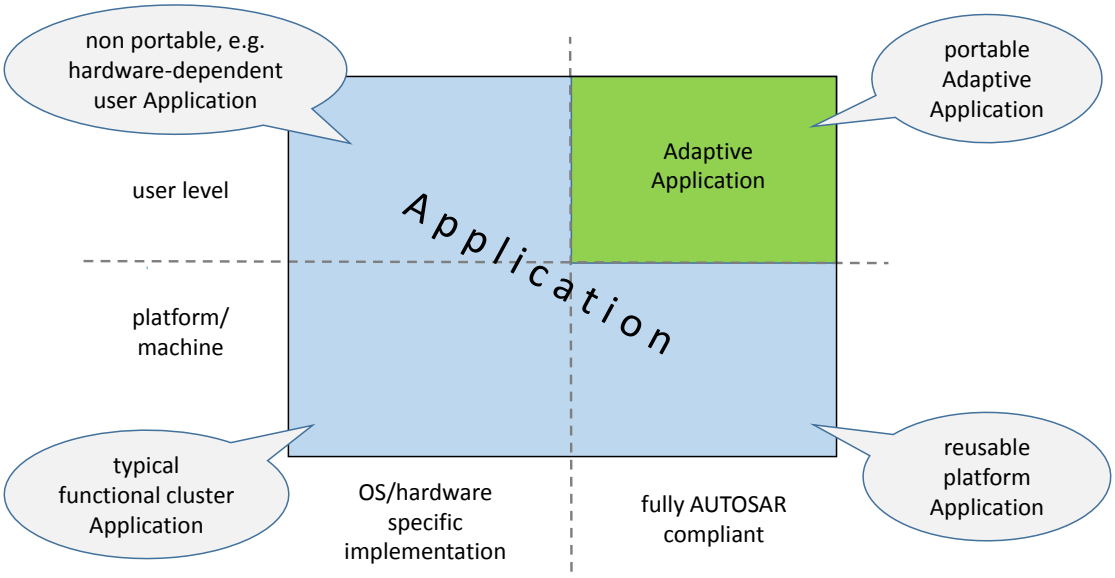

Figure 7.1: Types of Applications

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 Applications need to be agreed on beforehand. For details see [9].

7.1.4 Executable

An Executable is a software unit which is part of an Application. It has exactly one entry point (main function) [SWS_OSI_01300]. An Application can be implemented in one or more Executables [TPS_MANI_01008].

The lifecycle of Executables 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.	Execution Manifest, see 7.1.6 and [4], and Service Instance Manifest (not used by Execution Management).
Deployment and Removal	Binary installed on the target Machine. Previous version (if any) removed.	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.

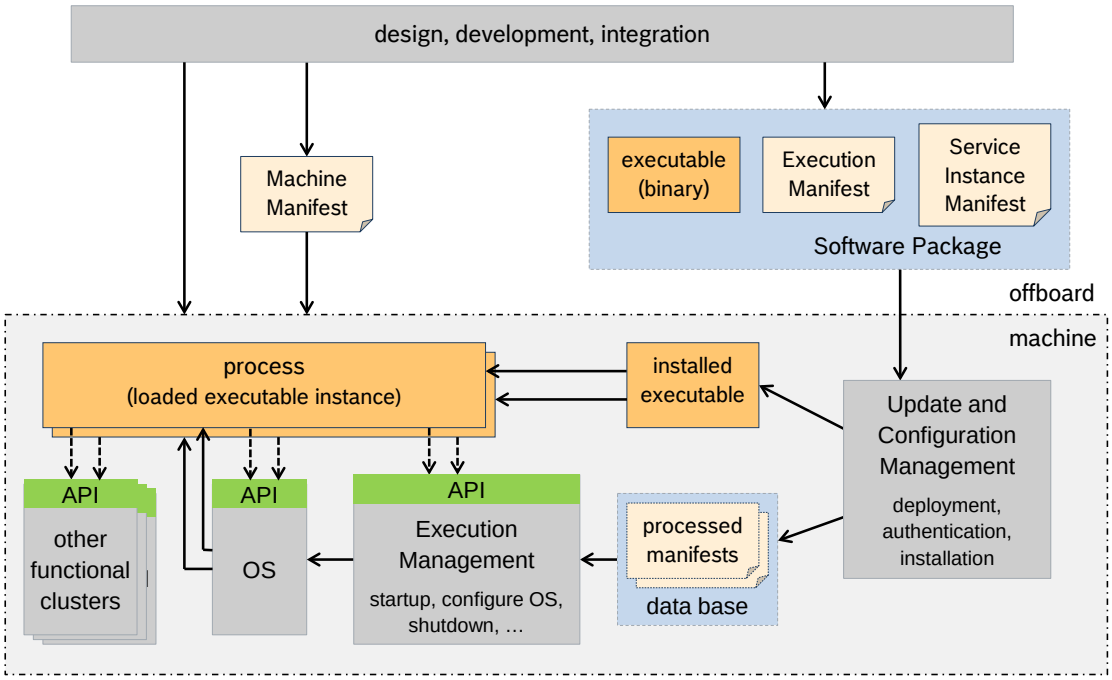

Figure 7.2: Executable Lifecycle from deployment to execution

7.1.5 Process

A Process is a started instance of an Executable. On the AUTOSAR Adaptive Platform, a Process technically is a POSIX process. For details on how Execution Management starts and stops Processes see 7.3.

Execution Management treats all Executables and the derived Processes the same way, independent of Application boundaries.

Remark: In this release of this document it is mostly assumed that Processes are self-contained, i.e. that they take care of controlling thread creation and scheduling by calling APIs of the Operating System Interface 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 providing State Management mechanisms (see 7.4) or APIs to support Deterministic Execution (see 7.6.3).

7.1.6 Execution Manifest

An Execution Manifest is created together with a Service Instance Manifest (not used by Execution Management) at design time and deployed onto a Machine together with the Executable it is attached to.

The Execution Manifest specifies the deployment related information of an Executable and describes in a standardized way the machine-specific configuration of Process properties (startup parameters, resource group assignment, scheduling priorities etc.).

The Execution Manifest is bundled with the actual executable code in order to support the deployment of the executable code onto the Machine.

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 Section 7.4 and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01014], [TPS_MANI_01015], [TPS_MANI_01059], [TPS_MANI_01017] and [TPS_MANI_01041]).

To perform its necessary actions, Execution Management imposes a number of requirements on the content of the Execution Manifest.

For more information regarding the Execution Manifest specification please see [4].

7.1.7 Machine Manifest

The Machine Manifest is also created at integration time for a specific Machine and is deployed like Execution Manifests whenever its contents change. The

Machine Manifest holds all configuration information which cannot be assigned to a specific Executable or its instances (the Processes), i.e. which is not already covered by an Execution 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 [4].

7.1.8 Manifest Format

The Execution Manifests and the Machine Manifest can be transformed from the original standardized ARXML 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 Update and Configuration Management) at installation time.

7.2 Execution Management Responsibilities

Execution Management is responsible for all aspects of Process execution management. A Process is a loaded instance of an Executable, which is part of an Application.

Execution Management is started as part of the AUTOSAR Adaptive Platform startup phase and is responsible for starting and terminating Processes.

Execution Management determines when, and possibly in which order, to start or stop Processes, i.e. instances of the deployed Executables, based on information in the Machine Manifest and Execution Manifests.

[SWS_EM_01030]{DRAFT} **Start of Process execution** ⌈ Execution Management shall initiate execution of a Process in such a way that the Process cannot start other Processes. ⌋*(RS_EM_00009, RS_AP_00131)*

Depending on the Machine State or on any other Function Group State, deployed Executables are started during AUTOSAR Adaptive Platform startup or later, however it is not expected that all will begin active work immediately since many Processes will provide services to other Processes and therefore wait and "listen" for incoming service requests.

Execution Management derives an ordering for startup/shutdown of deployed Executables within the context of Machine and/or Function Group State changes based on declared Execution Dependencies [SWS_EM_01050]. The dependencies are described in the Execution Manifests, see [TPS_MANI_01041].

Execution Management is **not** responsible for run-time scheduling of Processes since this is the responsibility of the Operating System [RS_OSI_01003]. However, Execution Management is responsible for initialization / configuration of the OS to enable it to perform the necessary run-time scheduling and resource management based on information extracted by Execution Management from the Machine Manifest and Execution Manifests.

7.3 Process Lifecycle Management

7.3.1 Execution State

Execution States characterizes the internal lifecycle of a Process. In other words, they describe it from the point of view of a Process that is executed. The states visible to the Process are defined by the ExecutionState enumeration.

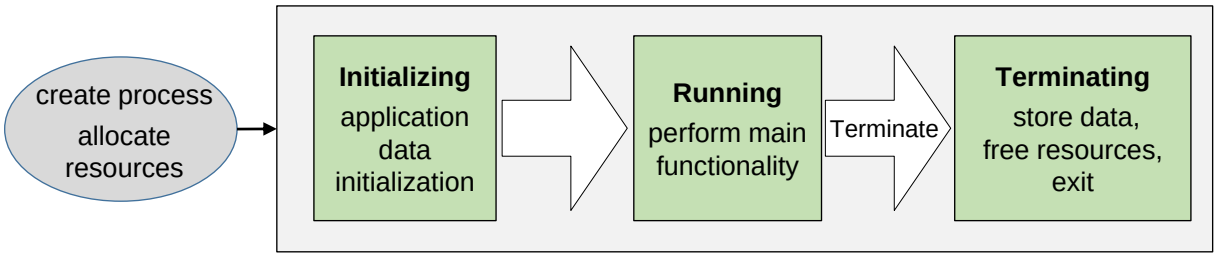

Figure 7.3: Execution States

[SWS_EM_01053]{DRAFT} **Execution State Running** ⌈ Execution Management shall consider Process initialization complete when the state kRunning is reported. ⌋*(RS_EM_00103, RS_AP_00131)*

Please note that Service Discovery can introduce non-deterministic delays and thus is advised to be done after reporting kRunning state. This implies that the Process may not have completed all its initialization when the kRunning state is reported by the Process, using the ExecutionClient::ReportExecutionState interface.

[SWS_EM_01055]{DRAFT} **Initiation of Process termination** ⌈ Execution Management shall initiate Process termination by sending the SIGTERM signal to the Process. ⌋*(RS_EM_00103, RS_AP_00131)*

Note that from the perspective of Execution Management, requirement [SWS_EM_01055] only requests the initiation of the steps necessary for termination. On receipt of SIGTERM, a Process acknowledges the request (by reporting the new state to Execution Management using the ExecutionClient::ReportExecutionState interface) and then commences the actual termination.

[SWS_EM_01070]{DRAFT} **Acknowledgement of termination request** ⌈ On reception of SIGTERM, the Process shall acknowledge the state change request by reporting kTerminating to Execution Management. ⌋*(RS_EM_00103, RS_AP_00131)*

[SWS_EM_01071]{DRAFT} **Initiation of Process self-termination** ⌈ A Process shall initiate self-termination by reporting the kTerminating state to Execution Management. ⌋*(RS_EM_00103, RS_AP_00131)*

During the Terminating state, the Process is expected to save persistent data and free all internally used resources. The Process indicates completion of the Terminating state by simply exiting (with an appropriate exit code).

Execution Management does not require an explicit notification of actual Process termination by the process itself as this would introduce a race condition. Instead, Execution Management as the parent Process can detect termination of the child Process and take the appropriate platform-specific actions such as processing execution dependencies that rely on the Terminated state and thus ensure that there is no overlap between these Processes when both are running.

Details on the response to "fault" error-codes, e.g. a non-zero exit code, will be defined in Section 7.8 in a future release of this document.

Correct *Execution State* reporting performed by Processes is a part of consistent behavior of Execution Management.

[SWS_EM_02243]{DRAFT} **Handling Execution State Running** ⌈ After Process creation, Execution Management shall ignore duplicate *Execution State* Running reports triggered by a specific Process. ⌋*(RS_EM_00103, RS_AP_00131)*

[SWS_EM_02244]{DRAFT} **Handling Execution State Terminating** ⌈ After initiation of Process termination, Execution Management shall ignore duplicate *Execution State* Terminating reports and inconsistent backward *Execution State* Running transition reports triggered by a specific Process. ⌋*(RS_EM_00103, RS_AP_00131)*

7.3.2 Process States

Process States characterize the lifecycle of a Process from the point of view of Execution Management. In other words, they represent Execution Management internal tracking of the *Execution States* (see Section 7.3.1). Note that each Process is independent and therefore has its own *Process State*.

[SWS_EM_01002]{DRAFT} **Idle Process State** ⌈ The **Idle** Process State shall be the Process State prior to creation of the Process and to resource allocation. ⌋ *(RS_EM_00103, RS_AP_00131)*

[SWS_EM_01003]{DRAFT} **Starting Process State** ⌈ The **Starting** Process State shall apply when the Process has been created and resources have been allocated. ⌋*(RS_EM_00103, RS_AP_00131)*

[SWS_EM_01004]{DRAFT} **Running Process State** ⌈ The **Running** Process State shall apply to a Process after it has been scheduled and it has reported kRunning Execution State to Execution Management. ⌋*(RS_EM_00103, RS_AP_00131)*

[SWS_EM_01005]{DRAFT} **Terminating Process State** ⌈ The **Terminating** Process State shall apply either from Execution Management sending SIGTERM signal to the Process, or after the Process has decided to self-terminate and informed Execution Management by reporting kTerminating Execution State. ⌋ *(RS_EM_00103, RS_EM_00011, RS_AP_00131)*

The kTerminating and kRunning Execution State indications from Process to Execution Management use the ExecutionClient::ReportExecutionState API (see Section 8.2.1.3).

[SWS_EM_01006]{DRAFT} **Terminated Process State** ⌈ The **Terminated** Process State shall apply after the Process has terminated and the Process resources have been freed. ⌋*(RS_EM_00103, RS_AP_00131)*

For [SWS_EM_01006], Execution Management observes the exit status of all Processes. The mechanism is implementation dependent but could, for example, use the POSIX waitpid() API.

From the resource allocation point of view, the **Terminated** Process State is similar to the **Idle** Process State – there is no Process running and no resources are allocated. However from the execution point of view, the **Terminated** Process State is different from **Idle** as it tells Execution Management that the Process has already been executed, terminated and can be now restarted (if needed) as specified in [SWS_EM_01066].

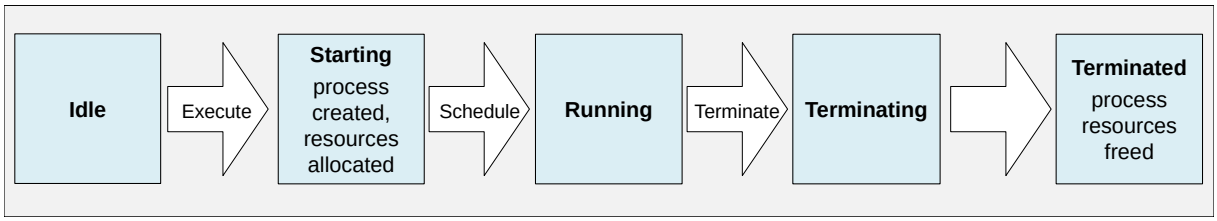

Figure 7.4: Process Lifecycle

7.3.3 Startup and Shutdown

7.3.3.1 Ordering

Execution Management can derive an ordering for the startup and shutdown of Processes within State Management framework based on the declared Execution Dependencies.

An Execution Dependency defines the provider of functionality required by a Process necessary for that Process to provide its own functionality. Execution Management ensures the dependent Processes are in the state defined by the Execution Dependency before the Process defining the dependency is started.

When considering dependencies on Services it is tempting to use Execution Dependencies. However general Service Discovery is a better mechanism as services can go ON or OFF at any time and the Processes are expected to cope with this situation. Please note this doesn't mean that Execution Dependencies can't be used to reduce service discovery delays.

Execution Dependencies are described in the Execution Manifests [TPS_MANI_01041].

Example 7.1

Consider a Process, $DataLogger$, which has an Execution Dependency on another Process, $Storage$. For startup this means $DataLogger$ has a Execution Dependency on $Storage$ so the latter is required to be started by Execution Management before $DataLogger$ so that $DataLogger$ can store its data.

[SWS_EM_01050]{DRAFT} **Start Dependent Processes** ⌈ During startup, Execution Management shall respect Execution Dependencies by ensuring that any Processes upon which the Process to be started depends have reached the requested Process State before starting the Process. ⌋*(RS_EM_00100, RS_AP_00131)*

The same Execution Dependencies used to define the startup order are also used to define the shutdown order. However the situation is reversed as Execution Management is required to ensure that dependent processes are shutdown **after** the process to ensure that the services required remain available until no longer required.

[SWS_EM_01051]{DRAFT} **Shutdown Processes** ⌈ During shutdown, Execution Management shall respect Execution Dependencies by ensuring that any Processes upon which the Process to be shutdown depends are not terminated before shutting down the Process. ⌋*(RS_EM_00100, RS_AP_00131)*

Example 7.2

Consider the same Process, $DataLogger$, as above which has an Execution Dependency on another Process, $Storage$. For shutdown the Execution Dependency indicates Execution Management is required to only shutdown $Storage$ after $DataLogger$ so the latter can flush its data during shutdown.

Note that [SWS_EM_01051] merely requires Execution Management to not terminate the dependent process(s) before shutting down a process. It is not an error if the Process has self-terminated so is not available to be terminated.

If no Execution Dependencies are specified between two Processes then no order is imposed and they can be started or shutdown in an arbitrary order.

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 Execution Manifest.

Execution Management parses the information and uses it to build the startup sequence to ensure that the required antecedent Processes have reached a certain *Process State* before starting a dependent Process [SWS_EM_01050].

7.3.3.2 Arguments

Execution Management provides argument passing for a Process containing one or more StateDependentStartupConfig in the role Process.stateDependentStartupConfig. This permits different Processes to be started with different arguments.

[SWS_EM_01012]{DRAFT} **Process Argument Passing** ⌈ At the initiation of startup of a Process, the aggregated StartupOptions of the StartupConfig referenced by the StateDependentStartupConfig shall be passed to the call of the exec-family based POSIX interface to start the Process by the Operating System based on [SWS_EM_01072], [SWS_EM_01073], [SWS_EM_01074], [SWS_EM_01075], [SWS_EM_01076] and [SWS_EM_01077]. ⌋*(RS_EM_00010, RS_AP_00131)*

The first argument on the command-line passed by Execution Management is the name of the Executable.

[SWS_EM_01072]{DRAFT} **Process Argument Zero** ⌈ Argument 0 shall be set to name of the Executable. ⌋*(RS_EM_00010, RS_AP_00131)*

Execution Management supports simple arguments that are passed directly to the Process without any additional processing. Please note that StartupOption.optionName will not be passed to the Process if StartupOption.optionKind = commandLineSimpleForm and can be omitted.

[SWS_EM_01073]{DRAFT} **Simple Arguments** ⌈ For each aggregated StartupOption at position n with StartupOption.optionKind = commandLineSimpleForm the nth argument shall be StartupOption.optionArgument. ⌋*(RS_EM_00010, RS_AP_00131)*

Execution Management supports short form arguments which are typically single characters. All short form arguments begin with a single dash (-) which is not included in the StartupOption.optionName.

[SWS_EM_01074]{DRAFT} **Short form arguments with option value** ⌈ For each aggregated StartupOption at position n with StartupOption.optionKind = commandLineShortForm and with multiplicity of StartupOption.optionArgument $= 1$ the nth argument shall be '-' + StartupOption.optionName + '␣' + StartupOption.optionArgument ⌋*(RS_EM_00010, RS_AP_00131)*

Note that requirement [SWS_EM_01074] includes the specification of mandatory whitespace; this is indicated by '␣' in the requirement text.

[SWS_EM_01075]{DRAFT} **Short form Arguments without option value** ⌈ For each aggregated StartupOption at position n with StartupOption.optionKind = commandLineShortForm and with multiplicity of StartupOption.optionArgument $= 0$ the nth argument shall be '-' + StartupOption.optionName ⌋ *(RS_EM_00010, RS_AP_00131)*

Execution Management supports long form arguments which are typically more meaningful to the user than short-form arguments. To distinguish long form arguments

from short form the former begin with a double dash (--) which is not included in the StartupOption.optionName.

[SWS_EM_01076]{DRAFT} **Long form Arguments with option value** ⌈ For each aggregated StartupOption at position n with StartupOption.optionKind = commandLineLongForm and with multiplicity of StartupOption.optionArgument $= 1$ the nth argument shall be '--' + StartupOption.optionName + '=' + StartupOption.optionArgument ⌋*(RS_EM_00010, RS_AP_00131)*

[SWS_EM_01077]{DRAFT} **Long form Arguments without option value** ⌈ For each aggregated StartupOption at position n with StartupOption.optionKind = commandLineLongForm and with multiplicity of StartupOption.optionArgument $= 0$ the nth argument shall be '--' + StartupOption.optionName ⌋*(RS_EM_00010, RS_AP_00131)*

7.3.3.3 Environment Variables

Execution Management initializes environment variables for Processes. Process specific environment variables are configured in its Execution Manifest. Machine specific environment variables are configured in the Machine Manifest. During runtime environment variables are accessible via POSIX `getenv()` command.

[SWS_EM_02246]{DRAFT} **Process specific Environment Variables** ⌈ Execution Management shall prepare environment variables based on the configuration from Process.stateDependentStartupConfig.startupConfig.environmentVariable and pass them during a process start. ⌋*(RS_EM_00010, RS_AP_00130, RS_AP_00131)*

[SWS_EM_02247]{DRAFT} **Machine specific Environment Variables** ⌈ Execution Management shall prepare environment variables based on the configuration from Machine.environmentVariable and pass them during a process start. ⌋*(RS_EM_00010, RS_AP_00130, RS_AP_00131)*

Please note that AUTOSAR meta model (see [4]) uses TagWithOptionalValue for environment variables definition ([TPS_MANI_01208] and [TPS_MANI_01209]). As explained there, the value (TagWithOptionalValue.value) can be omitted as a way of specifying environment variable with empty value.

[SWS_EM_02249]{DRAFT} **Missing value from Environment Variable definition** ⌈ Whenever Execution Manifest finds environment variable definition, that have TagWithOptionalValue.value missing, it should use empty string as a value for this environment variable. ⌋*(RS_EM_00010, RS_AP_00130, RS_AP_00131)*

[SWS_EM_02248]{DRAFT} **Environment Variables precedence** ⌈ Whenever the same environment variable is configured within both the Execution Manifest and the Machine Manifest then Execution Management shall use the environment variable value from the Execution Manifest. ⌋*(RS_EM_00010, RS_AP_00130, RS_AP_00131)*

7.3.4 Startup Sequence

When the Machine is started, the OS will be initialized first and then Execution Management is launched as one of the OS's initial Processes[1]. Other functional clusters and platform-level Applications of the Adaptive Platform Foundation and Adaptive Platform Services are then launched by Execution Management. After the Adaptive Platform Foundation and Adaptive Platform Services are up and running, Execution Management continues to launch user-level Applications.

Please note that an Application consists of one or more Executables. Therefore to launch an Application, Execution Management starts Processes as instances of each Executable.

[SWS_EM_01000]{DRAFT} **Startup order** ⌈ The startup order of the platform-level Processes shall be determined by Execution Management based on Machine Manifest and Execution Manifest information. ⌋*(RS_EM_00100, RS_AP_00131)*

Please see Section 7.1.6.

Figure 7.5 shows the overall startup sequence.

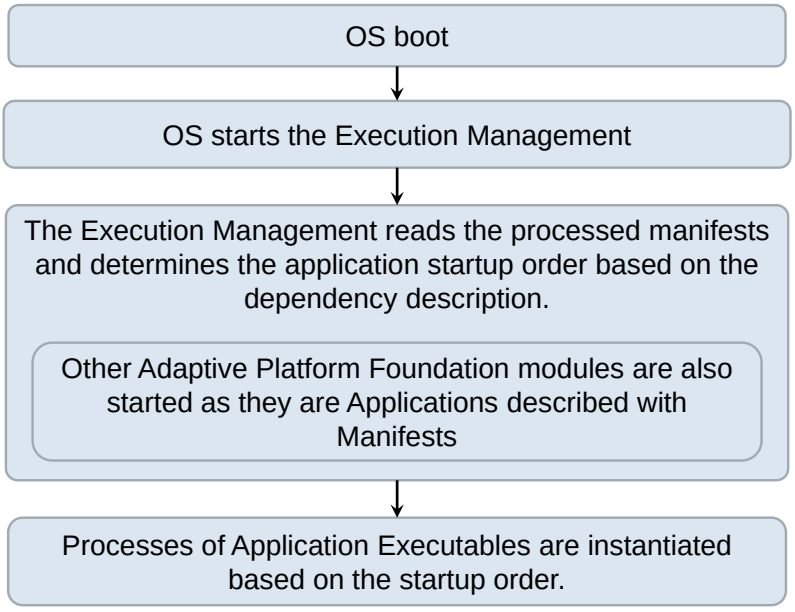

Figure 7.5: Startup sequence

[1] Typically the *init* process

7.3.4.1 Execution Dependency

Execution Management provides support to the AUTOSAR 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.

The Execution Dependencies, see [TPS_MANI_01041] and [TPS_MANI_1606] , are configured in the Execution Manifests, which are created at integration time based on information provided by the Application developer.

User-level Applications are expected to use the service discovery mechanisms of Communication Management as the primary mechanism for execution sequencing as this is supported both within a Machine and across Machine boundaries. Thus user-level applications should not rely on Execution Dependencies unless strictly necessary. Which Processes are running depends on the current Function Group States, including the Machine State, see Section 7.4. The integrator should ensure that all service dependencies are mapped to State Management configuration, i.e. that all dependent Processes are running when needed.

In real life, specifying a simple dependency to a Process might not be sufficient to ensure that the depending service is actually provided. Since some Processes shall reach a certain *Execution State* (see Section 7.3.1) to be able to offer their services to other Processes, the dependency information shall also refer to *Process State* of the Process specified as dependency. With that in mind, the dependency information may be represented as a pair like: <Process>.<ProcessState>. For more details regarding the *Process States* refer to Section 7.3.2.

The following dependency use-cases have been identified:

Dependency on `Running` Process State In case Process B has a simple dependency on Process A, the *Running Process State* of Process A is specified in the dependency section of Process B's Execution Manifest.

When Process B has a Running Execution Dependency to Process A, then Process B will only be started once the Process A reports Running state to the EM.

Dependency on `Terminated` Process State In case Process D depends on Self-terminating Process C, the *Terminated Process State* of Process C is specified in the dependency section of Process D's Execution Manifest.

If Process D has Terminated Execution Dependency on Process C, then Process D will only be started once Process C reaches the Terminated state.

If a Terminated Execution Dependency is specified on a non self-terminating Process then it will, by definition, time-out as the mentioned Process will not terminate until the next Function Group transition.

Note: No use-case has been identified for an Execution Dependency on other Process States, i.e. Idle or Terminating, and therefore these are not supported for Execution Dependency configuration.

Version information within the Execution 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. This also applies to the Processes which are instantiated from these Executables. An example for the definition of the version information attached to several Executables can be found in Listing Figure 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 Execution Dependency information can be found in Listing Figure 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>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
                DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVR/
                ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
                PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
                ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-
                MODE-DECLARATION-REF>
```

```
          </PROCESS-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
                DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVL/
                ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
                PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
                ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-
                MODE-DECLARATION-REF>
          </PROCESS-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
                DECLARATION-GROUP-PROTOTYPE">/Processes/Diag/
                ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
                PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
                ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-
                MODE-DECLARATION-REF>
          </PROCESS-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 Execution Management 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.

[SWS_EM_01400]{DRAFT} **Execution Dependency resolution** ⌈ When resolving Execution Dependencies, Execution Management shall not attempt to start any Processes that are not part of the same Function Group State as the depending Process. ⌋*(RS_EM_00100, RS_AP_00131)*

[SWS_EM_01001]{DRAFT} **Execution Dependency error** ⌈ If Execution Management needs to start Process A that depends on another Process B and Process B is not part of the same Function Group State as Process A, then Execution Management shall return an error (and fail to start Process A). ⌋*(RS_EM_00100, RS_AP_00131)*

Example 7.3

Let assume that Process “A” depends on the *Running Process 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 Dependency between the

two Processes, Process "A" would never start running in the new Machine State because it waits forever for Process "B". This is considered to be a configuration error and shall also cause run time error.

Please note that requirements [SWS_EM_01400] and [SWS_EM_01001] effectively forbids any Execution Dependencies that spans outside of a single Function Group State (or a Machine State) definition. This is done on purpose, as this kind of dependencies will introduce hidden dependencies between Function Groups and they will not be visible to State Management. If dependencies between Function Groups needs to be expressed (e.g. mapping software could have dependency on GPS software), then this should be done inside State Management. For more information see [10].

7.4 State Management

7.4.1 Overview

State Management functional cluster defines the operational state of an AUTOSAR Adaptive Platform, while Execution Management performs the transitions between different states.

The Execution Manifest allows to define in which states the Processes have to run (see [4]). As mentioned before, a Process is an instance of an Executable, which is part of an Application. State Management mechanisms grant full control over the set of Applications to be executed and ensures that Processes are only executed (and hence resources allocated) when actually needed.

Four different states are relevant for Execution Management:

Execution State – An Execution States characterizes the internal lifecycle of each started Process, see Section 7.3.1

Process State – Process States are managed by an Execution Management internal state machine. For details see Section 7.3.2.

Machine State – see Section 7.4.2

Function Group State – see Section 7.4.3

An example for the interaction between these states will be shown in section Section 7.4.4.

7.4.2 Machine State

Execution Management requires that at least one Function Group will be configured for each Machine. This Function Group shall have the name "MachineState" and in this document, this particular Function Group State is named Machine State. Machine State is meant to represent a global state of Machine.

The Function Group "MachineState" has several mandatory states (see [SWS_EM_02250], [SWS_EM_01024] and [SWS_EM_01025]) that are also expected to be configured for each machine. Additional Machine States can be defined on a machine specific basis and are therefore not standardized.

Function Group States (including Machine States), define the current set of running Processes. Each Application can declare in its Execution Manifests in which Function Group States its Processes shall be running. A ModeDeclaration for each required Machine State has to be defined in the Machine Manifest [TPS_MANI_03066].

Machine States (as well as other Function Group States) are requested by State Management. The set of active states is significantly influenced by vehicle-wide events and modes. For details on state change management see Section 7.4.5.

[SWS_EM_01032]{DRAFT} **Machine States configuration** ⌈ Execution Management shall obtain configuration of the Function Group "MachineState" from Machine Manifest and set-up Machine States management. ⌋*(RS_EM_00101, RS_AP_00131)*

The start-up sequence from initial state Startup to the point where State Management, SM, requests the initial running machine state StateXYZ is illustrated in Figure 7.6.

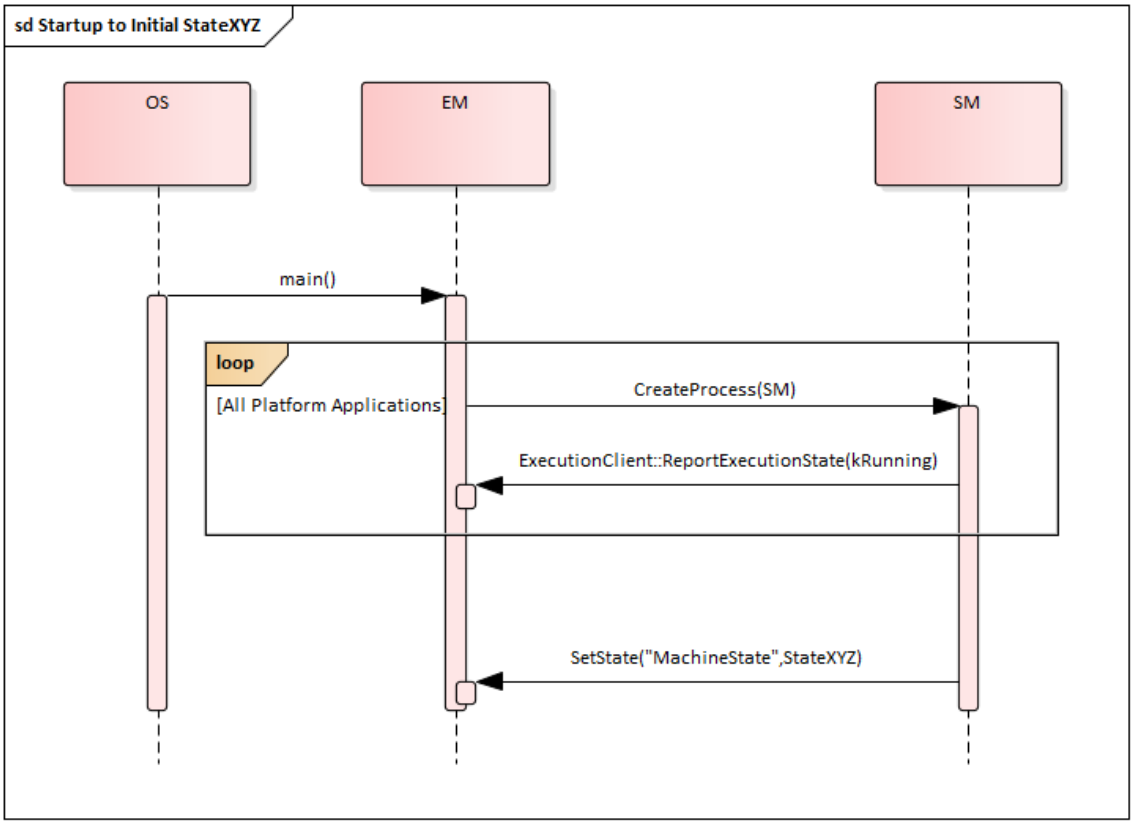

Figure 7.6: Start-up Sequence – from Startup to initial running state StateXYZ

An arbitrary state change sequence to machine state StateXYZ is illustrated in Figure 7.7. Here, on receipt of the state change request, Execution Management terminates running Processes and then starts Processes active in the new state before confirming the state change to State Management.

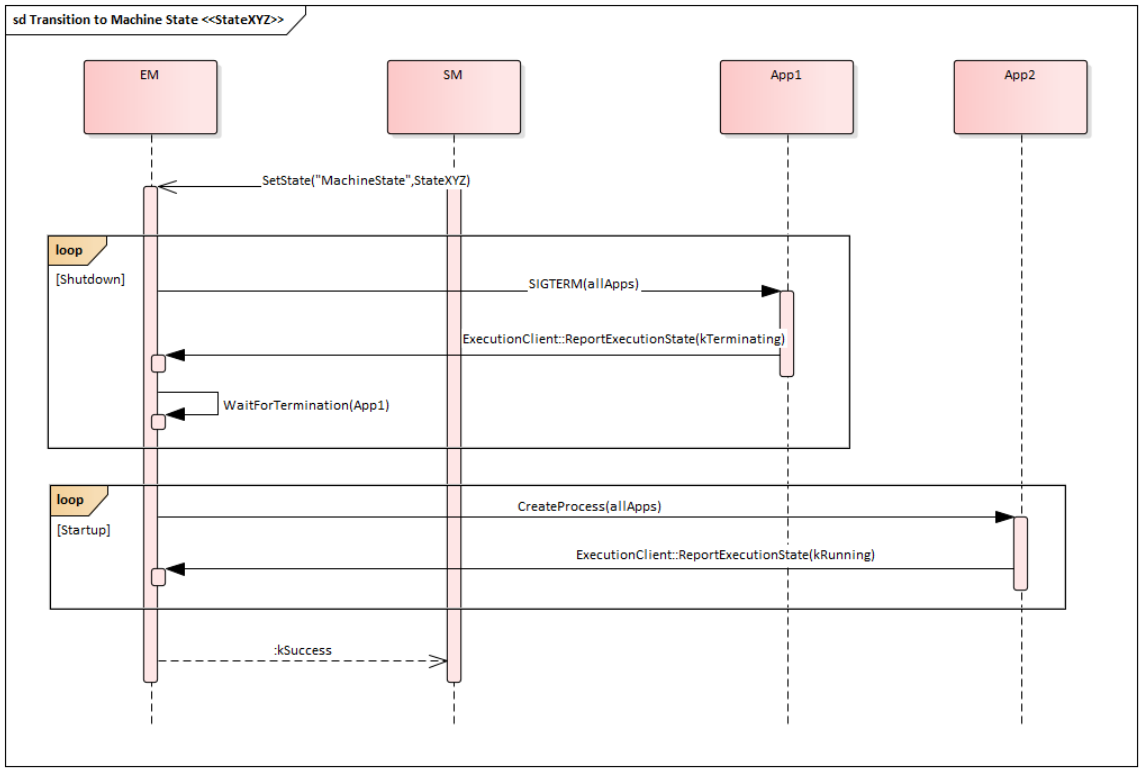

Figure 7.7: State Change Sequence – Transition to machine state `StateXYZ`

7.4.2.1 Startup

[SWS_EM_02250]{DRAFT} **Machine State Startup** ⌈ `Execution Management` shall ensure that `Startup` state is configured for a `Function Group` with name `"MachineState"`. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_01023]{DRAFT} **Self initiation of Machine State Startup transition** ⌈ `Execution Management` shall self initiate the state transition to the `Startup Machine State`, as soon as possible after the startup of `Execution Management`. ⌋ *(RS_EM_00101, RS_AP_00131)*

Please note that for `Machine State` transitions, the requirements of section Section 7.4.5 apply.

[SWS_EM_02241]{DRAFT} **Machine State Startup Completion** ⌈ Upon completion of `Machine State` transition to the `Startup` state, `Execution Management` shall notify the `State Management` that the `Startup` state of `Machine State` has been reached. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02242]{DRAFT} **Further Function Group State Changes** ⌈ `Execution Management` shall not self initiate any further `Function Group State` changes (this includes `Machine State`) by itself. ⌋*(RS_EM_00101, RS_AP_00131)*

Execution Management will be controlled by other software entities and should not execute any Function Group State changes on its own (with one exception: [SWS_EM_01023]). This creates some expectations towards system configuration. The specification expects that State Management will be configured to run in every Machine State (this includes Startup, Shutdown and Restart). Above expectation is needed in order to ensure that there is always a software entity that can introduce changes in the current state of the Machine. If (for example) system integrator doesn't configure State Management to be started in Startup Machine State, then Machine will never be able transit to any other state and will be stuck forever in it. This also applies to any other Machine State that doesn't have State Management configured.

7.4.2.2 Shutdown

Execution Management does not perform shutdown of the Operating System. Instead it is required that at least one Process provides a mechanism to shutdown the Operating System. This Process is expected to be configured to run inside Shutdown Machine State. See [4] [constr_1618].

[SWS_EM_01024]{DRAFT} **Machine State Shutdown** ⌈ Execution Management shall ensure that Shutdown state is configured for a Function Group with name "MachineState". ⌋*(RS_EM_00101, RS_AP_00131)*

A request to Execution Management to change the current Machine State to Shutdown is handled the same as any other Function Group state change request. From the point of view of Execution Management all Function Groups are independent and therefore changes to them, can be applied without any side effects. However, from the point of view of State Management, where knowledge of the dependencies between different Function Groups exist this may not be true. AUTOSAR assumes that State Management will requests Machine State Shutdown when it's valid to do so; see [10] for advice on how to orchestrate shutdown of the Machine.

As mentioned in Section 7.4.2.1 AUTOSAR assumes that State Management will be configured to run in Shutdown. State transition is not a trivial system change and it can fail for a number of reasons - in which case State Management should remain alive so you can report an error and wait for further instructions. Please note that very purpose of this state is to shutdown Machine (this includes State Management) in a clean manner. Unfortunately this means that at some point State Management will no longer be available to report errors and subsequent errors should be handled through implementation specific mechanisms.

7.4.2.3 Restart

Execution Management does not perform restart of the Operating System. To restart the system it is required that at least one Process provide a mechanism to restart the Operating System. This Process is expected to be be configured to run inside Restart Machine State. See [4] [constr_1619].

[SWS_EM_01025]{DRAFT} **Machine State Restart** ⌈ Execution Management shall ensure that Restart state is configured for a Function Group with name "MachineState". ⌋*(RS_EM_00101, RS_AP_00131)*

From the point of view of Execution Management, the Restart state of a Function Group with name "MachineState" is very similar to a Shutdown state. For the reasons mentioned in Section 7.4.2.2, a state transition to Restart is handled the same as any other Function Group state transition; please see [10] for advice on how to orchestrate restart of the Machine.

As mentioned in Section 7.4.2.1 AUTOSAR assumes that State Management will be configured to run in Restart. The reasons for doing so are the same as for Section 7.4.2.2.

7.4.3 Function Group State

If there is a group of functionally coherent Applications installed on the machine, it will be useful to have ability of controlling them together. For that very reason the concept of Function Groups was introduced to AUTOSAR Adaptive Platform.

Each Function Group has its own set of Processes and set of states called Function Group States. Each Function Group State defines which Processes shall be started when State Management requests Function Group State activation from Execution Management. Please note that minimal size of a Function Group is one Process and maximum size is implementation limited.

The Function Groups mechanism is very flexible and is intended as a tool used to start and stop Processes of Applications. System integrator can assign Processes to a Function Group State and then request it by State Management. For details on state change management see Section 7.4.5.

In general, Machine States (see Section 7.4.2) are used to control machine lifecycle (startup/shutdown/restart) and Processes of platform level Applications, while other Function Group States individually control Processes which belong to groups of functionally coherent user level Applications. Please note that this doesn't mean that all Processes of platform level Applications has to be controlled by Machine States.

Figure 7.8 shows an example of state change sequence where several `Processes` reference `Machine States` and `Function Group States` of two additional `Function Groups` **FG1** and **FG2**. For simplicity, only the three static Process States `Idle`, `Running`, and `Terminated` are shown for each process.

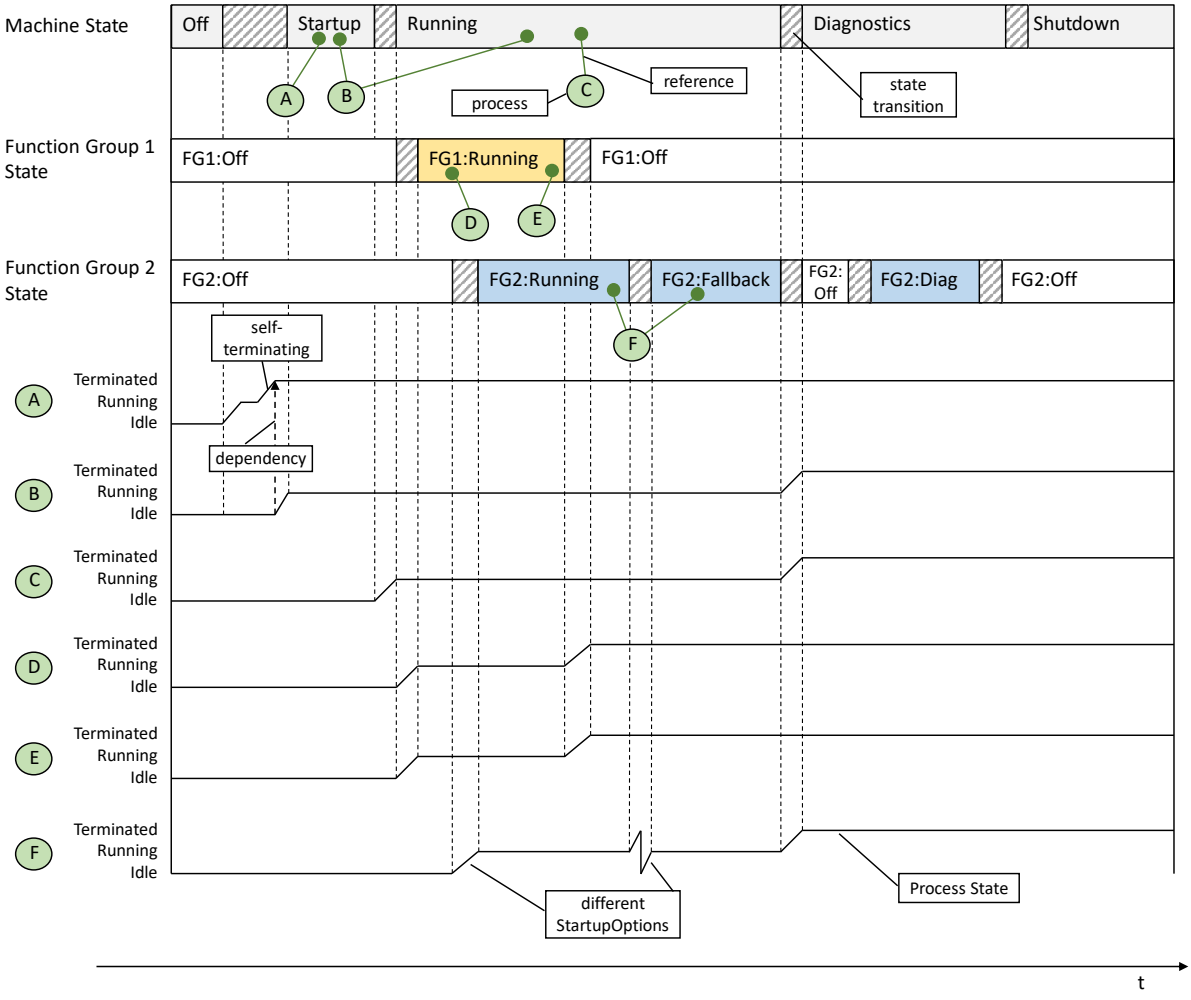

Figure 7.8: State dependent process control

- `Process` **A** references the `Machine State` `Startup`. It is a `Self-terminating Process`, i.e. it terminates after executing once.
- `Process` **B** references `Machine States` `Startup` and `Running`. It depends on the termination of `Process` **A**, i.e. an `Execution Dependency` has been configured, as described in Section 7.3.4.1
- `Process` **C** references `Machine State` `Running` only. It terminates when `Machine State` `Diagnostics` is requested by `State Management`.

- Processes **D** and **E** references Function Group State FG1:Running only and there is no Execution Dependency configured between them. Execution Management will start and terminate them in an arbitrary order (e.g. in parallel if possible).
- 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 should 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 should be considered.

[SWS_EM_01107]{DRAFT} **Function Group configuration** ⌈ Execution Management shall obtain configuration of the Function Group from the Machine Manifest to set-up the Function Group specific state management. ⌋*(RS_EM_00101, RS_AP_00131)*

A proper system configuration requires that each Process references in its Execution Manifest one or more Function Group States (which can be Machine States) of the same Function Group.

[SWS_EM_01013]{DRAFT} **Function Group State** ⌈ Execution Management shall support the execution of a specific Process, depending on the current Function Group State and on information provided in the Execution Manifests. ⌋ *(RS_EM_00101, RS_AP_00131)*

Each Process is assigned to one or several startup configurations (StartupConfig), which each can define the startup behavior in one or several Function Group States (including Machine States). For details see [4]. By parsing this information from the Execution Manifests, Execution Management can determine which Processes need to be launched if a specific Function Group State is entered, and which startup parameters are valid.

[SWS_EM_01033]{DRAFT} **Process start-up configuration** ⌈ To enable a Process to be launched in multiple Function Group States, Execution Management shall be able to configure the Process start-up on every Function Group State change based on information provided in the Execution Manifest. ⌋ *(RS_EM_00009, RS_EM_00101, RS_AP_00131)*

[SWS_EM_01109]{DRAFT} **Misconfigured Process - not assigned to a Function Group** ⌈ In the event of a misconfigured system, Execution Management shall not start a Process that doesn't reference at least one state. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02254]{DRAFT} **Misconfigured Process - assigned to more than one Function Group** ⌈ In the event of a misconfigured system, Execution Management shall not start a Process that references states from more than one Function Group. ⌋*(RS_EM_00101, RS_AP_00131)*

Please note AUTOSAR doesn't support the possibility of assigning a single Process to more than one Function Group, see [4] ([constr_1688]).

[SWS_EM_01110]{DRAFT} **Off States** ⌈ Each Function Group (including the Function Group "MachineState") has an Off State which shall be used by Execution Management as default Function Group State, if no other state was requested. ⌋*(RS_EM_00101, RS_AP_00131)*

Please note that [SWS_EM_01110] and [SWS_EM_01023] together define the very first Function Group state transition after the power up. When Execution Management starts it performs Machine State transition from the "Off" state (the default state) to the "Startup" state.

Processes reference in their Execution Manifest the states in which they want to be executed. A state can be any Function Group State, including a Machine State. For details see [4], especially "State-dependent Startup Configuration" chapter and "Function Groups" chapter.

The arbitrary state change sequence as shown in Figure 7.7 applies to state changes of any Function Group - just replace "MachineState" by the name of the Function Group. On receipt of the state change request, Execution Management terminates no longer needed Processes and then starts Processes active in the new Function Group State before confirming the state change to State Management. For details see Section 7.4.5.

7.4.4 State Interaction

Figure 7.9 shows a simplified example for the interaction between different types of states, after State Management functional cluster has requested different Function Group States . One can see the state transitions of the Function Group and the Process and Execution States of one Process which references one state of this Function Group, ignoring possible delays and dependencies if several Processes were involved.

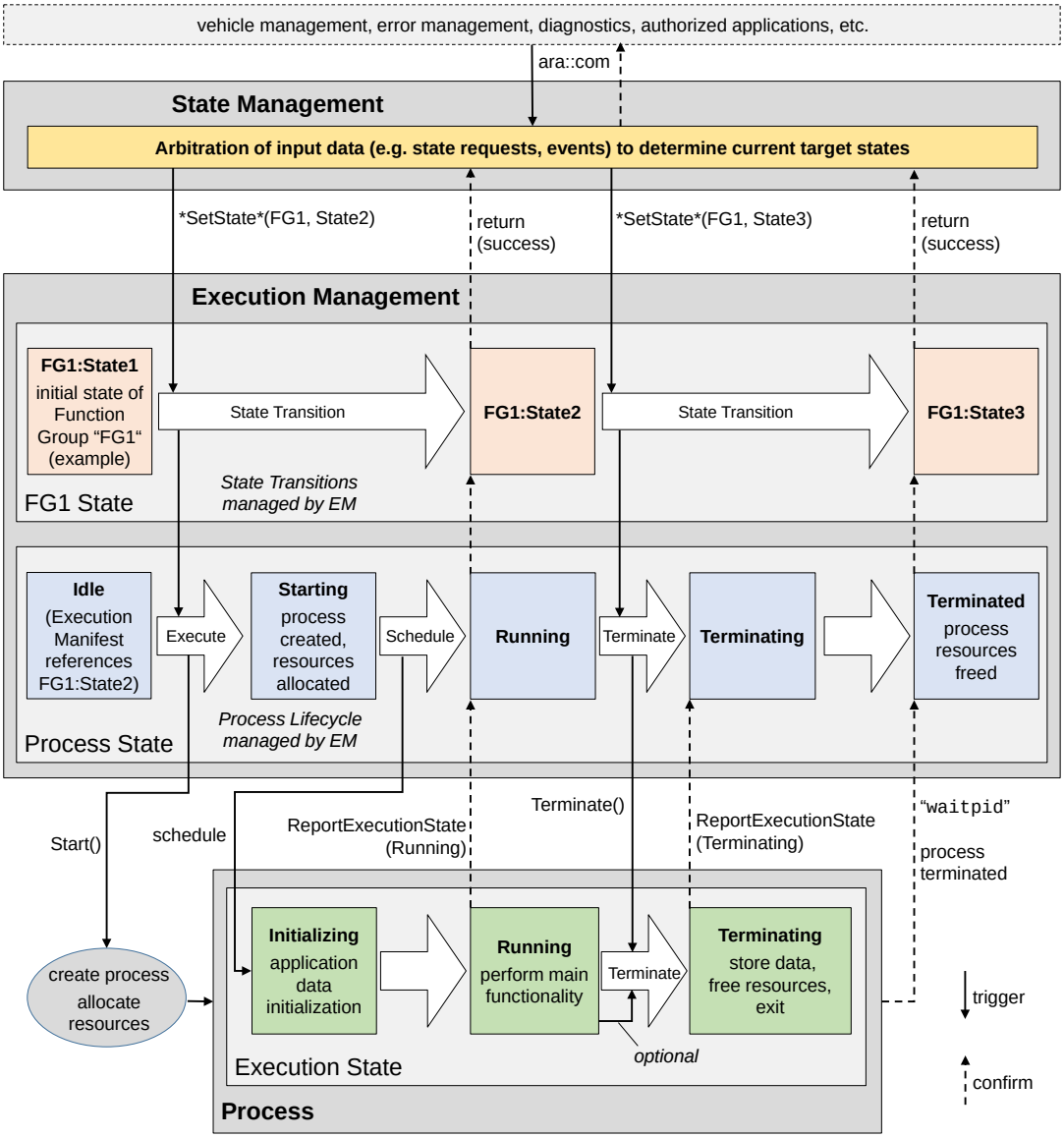

Figure 7.9: Interaction between states

7.4.5 State Transition

State Management can request to change one or several Function Group States (including the Machine State) from Execution Management by passing pairs of <Function Group><requested State> as parameters. In case of a Machine State change, the name of the Function Group is "MachineState".

[SWS_EM_01026]{DRAFT} **State Change** ⌈ A state change request by State Management shall lead to immediate state transitions and hereof a state change to the requested Function Group States. ⌋(*RS_EM_00101*, *RS_AP_00131*)

State Management can request multiple Function Group State changes sequentially by issuing the next state change request after the last one has been successfully finished, or atomically within the same state change request, which leads to multiple coherent state changes. However, the following restriction applies to avoid undefined behavior while the state transitions are performed by Execution Management:

[SWS_EM_01034]{DRAFT} **Deny State Change Request** ⌈ Execution Management shall deny state change requests, that are received before all previously requested Function Group State transitions are completed. If a request is denied, Execution Management shall return an error code to the requester of the state transition. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02058]{DRAFT} **State Transition Timeout** ⌈ If a timeout is detected when stopping or starting Processes at a state transition, Execution Management shall return an error code to State Management as the only user of the interface. ⌋ *(RS_EM_00101, RS_AP_00131)*

This implies that the state change request blocks until the state transitions are completed or until an error is detected.

[SWS_EM_02056]{DRAFT} **State Change Failed** ⌈ Execution Management shall return an error code to State Management when other or unspecified errors occur at a state transition. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02057]{DRAFT} **State Change Successful** ⌈ When Execution Management succeeds with the requested state transitions, a success code shall be returned to State Management. ⌋*(RS_EM_00101, RS_AP_00131)*

A table that summarized the requirements of this section can be found in Appendix Section B.2.1.

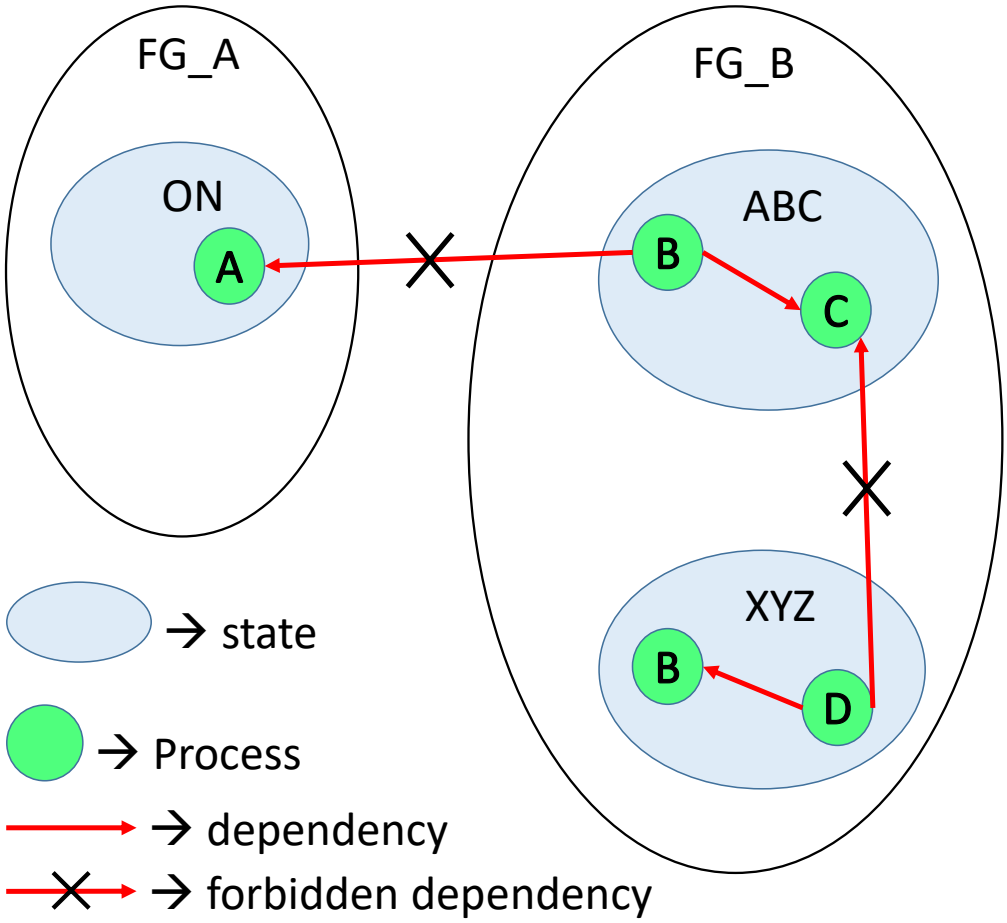

Figure 7.10: Example configuration for state transition

Before we specify how internals of a state transition works, let's consider an example configuration illustrated in figure Figure 7.10. As we can see `Execution Dependencies` that spans outside of a `Function Group` and moreover of a single `Function Group State` are forbidden. The dependency from `Process` **B** (inside `Function Group` *FG_B*) to `Process` **A** (inside `Function Group` *FG_A*) is forbidden, as it would introduce hidden dependencies between `Function Groups` that are not visible to `State Management`. If system configuration requires this kind of dependencies, please see [10] for advice on how to configure them. Dependencies outside of a single `Function Group State` definition are forbidden, as they would result in starting a `Process` that is not configured to run in the given `State`. For more information on `Execution Dependencies` see chapter Section 7.3.4.1 ([SWS_EM_01400] and [SWS_EM_01001]).

From the above we can conclude that each `Function Group` is a separate entity and state transition of one `Function Group` doesn't have side effects on another `Function Group`. Please note that this is true from the point of view of `Execution Management` and may differ from the point of view of `State Management` (see [10] if you need more information on this).

In the following requirements, the term "the Process references a State" means that a Process has in its Execution Manifest an aggregation of StateDependentStartupConfig in the role Process.stateDependentStartupConfig with an instanceRef to a ModeDeclaration in the role StateDependentStartupConfig.functionGroupState that belongs to that State.

CurrentState is the current (currently active) State, of a Function Group for which the state transition was requested; or the current Machine State if the Function Group has "MachineState" name. In short this is a Function Group State or Machine State.

RequestedState is the state that will become the CurrentState, once the state transition finishes successfully.

In other words CurrentState is the starting point of the transition, the list of the Processes that should be currently running inside the Function Group (please note the existence of Self-terminating Processes). RequestedState is a destination point of the state transition, the list of the Processes that will be running inside of the Function Group once the state transition finishes successfully (please note the existence of Self-terminating Processes).

StartupConfig it is a StateDependentStartupConfig that is aggregated in the role Process.stateDependentStartupConfig for a given Process.

State transition is a complicated process, however it is composed out of three simple logical steps:

- Terminate all Processes that are currently running and are not needed in the RequestedState
- Restart all Processes that are currently running and have StartupConfig that differs between the CurrentState and the RequestedState
- Start all Processes that are not running currently and are needed in the RequestedState

Please see Section 7.3.1 and Section 7.3.2 for more detail information on how Execution Management handles termination and start of Processes (restart is a sequence of termination and start).

[SWS_EM_01060]{DRAFT} **State transition - termination behavior** ⌈ On state transition Execution Management shall terminate all Processes that references the CurrentState in its Execution Manifest, but don't references the RequestedState and have Process State different than [Idle or Terminated]. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02251]{DRAFT} **State transition - restart behavior** ⌈ On state transition Execution Management shall terminate all Processes that references the CurrentState in its Execution Manifest, but references the RequestedState with different StartupConfig and have Process State different than [Idle or Terminated]. ⌋*(RS_EM_00101, RS_AP_00131)*

Please note that [SWS_EM_02251] only request a termination of Processes, the start part will fall under [SWS_EM_01066] requirement thus making the restart complete.

Execution Management monitors the time required by each Process to terminate. The default value of the Process termination timeout is defined by the system integrator in the Machine Manifest, see [TPS_MANI_03151]. This value may be overwritten for individual Processes by defining the Process termination timeout parameter in the Execution Manifest, see [TPS_MANI_03150].

[SWS_EM_01065]{DRAFT} **State transition - Process termination timeout monitoring** ⌈ Execution Management shall monitor the time required by the Process to terminate (the time needed by the Process to reach the Terminated Process State). ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02255]{DRAFT} **State transition - Process termination timeout reaction** ⌈ In case a Process termination timeout occurred, Execution Management shall request the Operating System to terminate the underlying process. ⌋*(RS_EM_00101, RS_AP_00131)*

On multi-process POSIX platforms, this could be achieved using a SIGKILL signal.

[SWS_EM_02252]{DRAFT} **State transition - Process termination timeout reporting** ⌈ When the termination of a Process resulted in the timeout, Execution Management shall perform following actions:

- Notify Platform Health Management about the timeout (using a Health Channel), to initiate appropriate recovery actions.
- Report error code back to State Management to indicate that the State change request cannot be fulfilled.

⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_01066]{DRAFT} **State transition - start behavior** ⌈ On state transition Execution Management shall start all Processes that references the RequestedState in its Execution Manifest and have Process State that is [Idle or Terminated]. ⌋*(RS_EM_00101, RS_AP_00131)*

Execution Management monitors the time required by each Process to start. The value of the Process start-up timeout is defined by the system integrator in the Execution Manifest, see [TPS_MANI_03149].

[SWS_EM_02253]{DRAFT} **State transition - Process start-up timeout monitoring** ⌈ Execution Management shall monitor the time required by the Process to start-up (the time needed by the Process to reach the Running Process State). ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02256]{DRAFT} **State transition - Process start-up timeout reaction** ⌈ In case a Process start-up timeout occurred, Execution Management shall request Process termination. If Process doesn't terminate as requested, Execution Management shall request the Operating System to terminate the underlying process. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_01068]{DRAFT} **State transition - Process start-up timeout reporting** ⌈ When the start-up of a Process resulted in the timeout, Execution Management shall perform following actions:

- Notify Platform Health Management about the timeout (using a Health Channel), to initiate appropriate recovery actions.
- Report error code back to State Management to indicate that the State change request cannot be fulfilled.

⌋*(RS_EM_00101, RS_AP_00131)*

When starting new Processes, Execution Management is obligated to perform dependency resolution. When doing so it may came across a configuration where Process B depends on Process A, but Process A needs to be restarted during state change. Please see Figure 7.11 for more details.

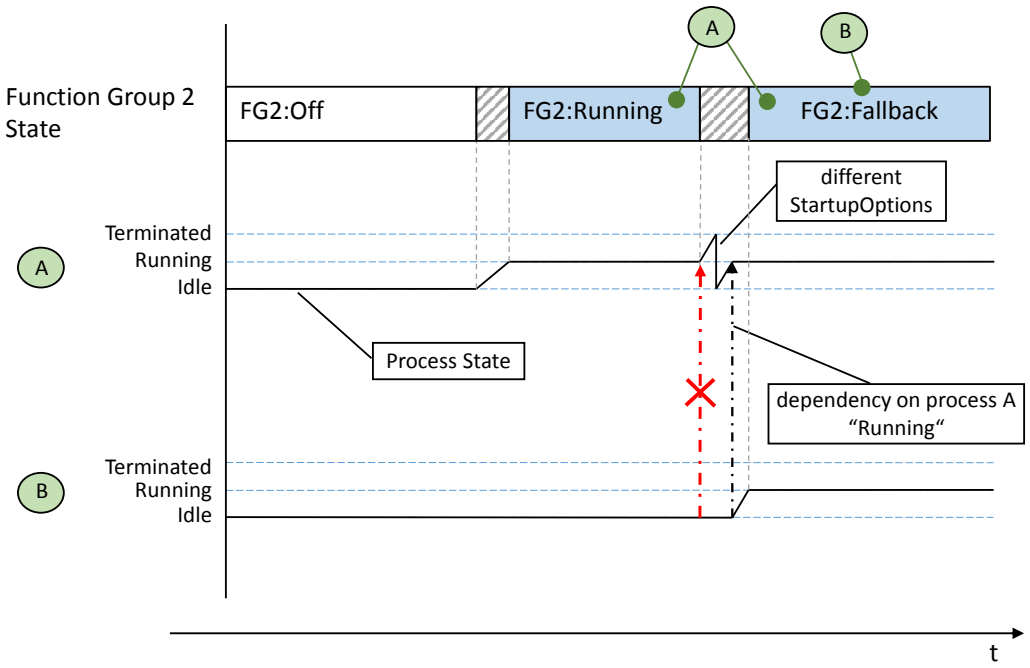

Figure 7.11: Dependency resolution during state change

[SWS_EM_02245]{DRAFT} **Dependency resolution during state change** ⌈ Execution Management shall ensure that Execution Dependency resolution is performed against the Processes that are configured for RequestedState. ⌋ *(RS_EM_00101, RS_AP_00131)*

Please note that [SWS_EM_02245] doesn't bring new functionality to state transition. It merely ensures that [SWS_EM_02251] and [SWS_EM_01066] are performed on Process A, before [SWS_EM_01066] is performed on Process B. If this order is not ensured then [SWS_EM_02245] could not be satisfied as Process A will be a Process that is configured for CurrentState and not for RequestedState.

[SWS_EM_01067]{DRAFT} **Finish of a successful state transition** ⌈ When all operation required for a state transition has been performed successfully, Execution Management shall consider the transition to be complete, set the CurrentState to the RequestedState and report success back to State Management. ⌋*(RS_EM_00101, RS_AP_00131)*

7.4.6 State Information

[SWS_EM_01028]{DRAFT} **Get State Information** ⌈ Execution Management shall provide an interface to retrieve the current Function Group State by passing a Function Group identifier as parameter. ⌋*(RS_EM_00101, RS_AP_00131)*

In case the current Machine State shall be retrieved, the identifier of the "MachineState" Function Group has to be used.

As well as potentially returning the requested state information the interface to retrieve the current Function Group State also returns information on whether or not the requested information can be provided. The possible responses are specified by [SWS_EM_02044], [SWS_EM_02049] and [SWS_EM_02050].

[SWS_EM_02044]{DRAFT} **State Change in Progress** ⌈ If Execution Management performs a state transition of the Function Group for which state information is requested, Execution Management shall return to the requester of the state information that it's busy and cannot provide a current state. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02049]{DRAFT} **State Change Failed** ⌈ If the last state change of the Function Group State, for which state information is requested, failed, then Execution Management shall return an error code to the requester of the state information. ⌋*(RS_EM_00101, RS_AP_00131)*

[SWS_EM_02050]{DRAFT} **State Information Success** ⌈ If Execution Management can successfully provide the requested state information, Execution Management shall return a success code to the requester of the state information. ⌋*(RS_EM_00101, RS_AP_00131)*

A table that summarized the requirements of this chapter can be found in Appendix Section B.2.2.

7.5 Application Recovery Actions

7.5.1 Overview

Execution Management is responsible for the state dependent management of Process start/stop, so it has to have the special right to start and stop Processes.

The Platform Health Management monitors Processes and could trigger a Recovery Action in case any Process behaves not within the specified parameters.

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

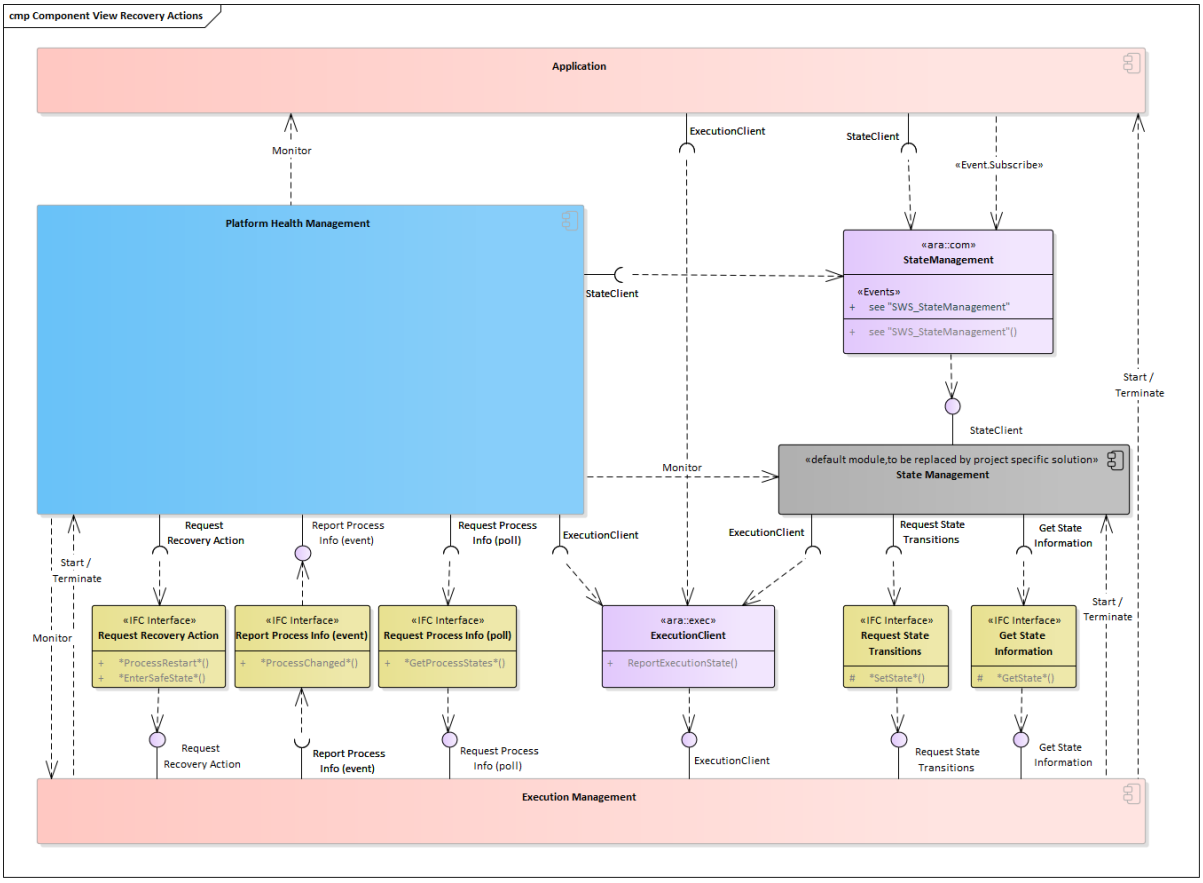

Figure 7.12: Adaptive Platform - Recovery Action Architecture

7.5.2 Process State Information

7.5.2.1 Get Process States Information

[SWS_EM_02076]{DRAFT} **Get Process States Information** ⌈ Execution Management shall provide an inter functional cluster interface for Platform Health

Management to receive a list of all currently running Processes. ⌋(*RS_EM_00013*, *RS_AP_00131*)

7.5.2.2 Process State Transition Event

[SWS_EM_02077]{DRAFT} **Process State Transition Event** ⌈ Execution Management shall call an inter functional cluster interface provided by Platform Health Management to report Process state changes. ⌋(*RS_EM_00013*, *RS_AP_00131*)

7.5.3 Recovery Actions

7.5.3.1 Process Restart

[SWS_EM_01016]{DRAFT} **Process Restart** ⌈ Execution Management shall provide an inter functional cluster interface to restart a specific Process on the request from the Platform Health Management. ⌋(*RS_EM_00013*, *RS_AP_00131*)

[SWS_EM_01062]{DRAFT} **Process Restart Behavior** ⌈ Execution Management shall restart a specific Process on the request from the Platform Health Management. ⌋(*RS_EM_00013*, *RS_AP_00131*)

[SWS_EM_01063]{DRAFT} **Process Restart Failed** ⌈ Execution Management shall return an error code to the requester of the Process restart when the Process restart could not be finished successfully. ⌋(*RS_EM_00013*, *RS_AP_00131*)

[SWS_EM_01064]{DRAFT} **Process Restart Successful** ⌈ When Execution Management succeeds with restarting the Process, a success code shall be returned to the requester of the Process restart. ⌋(*RS_EM_00013*, *RS_AP_00131*)

Remark: The Process Restart IFC API is a powerfull but also a critical API. The integrator should be sure about all consequences when using this API. Usually such an API could be used to restart the State Manager, all other scenarious might become complex.

7.5.3.2 Enter Safe State

[SWS_EM_01018]{DRAFT} **Enter Safe State** ⌈ Execution Management shall provide an inter functional cluster interface to force Execution Management to switch to specific Function Group States (which can include a Machine State) on the request from the Platform Health Management. ⌋(*RS_EM_00013*, *RS_AP_00131*)

[SWS_EM_01061]{DRAFT} **Enter Safe State Behavior** ⌈ An Enter Safe State request shall stop any currently "ongoing" state transition and process the transition to the Safe State. Execution Management shall ignore any incoming requests

for "process restart" recovery action from `Platform Health Management` and any state change request from `State Management` after "Enter Safe State" received. ⌋
(RS_EM_00013, RS_AP_00131)

7.6 Deterministic Execution

7.6.1 Determinism

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

In the context of Execution Management, 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.

It is also worthwhile to note that 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.

There are multiple elements in determinism and here we distinguish them as follows:

- Time Determinism: The output of the calculation is always produced before a given deadline (a point in time).
- Data Determinism: Given the same input and internal state, the calculation always produces the same output.
- Full Determinism: Combination of Time and Data Determinism as defined above.

In particular, deterministic behavior is 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.

Expected use cases of the AUTOSAR Adaptive Platform where such determinism is required include:

- Software Lockstep: To execute ASIL C/D applications with high computing performance demands, specific measures, such as software lockstep are required, due to high transient hardware error rates of high performance microprocessors. Software lockstep is a technique where the calculation is done redundantly through two different execution paths and the results are compared. To make the redundant calculations comparable, software lockstep requires a fully deterministic calculation. For details see 7.6.2.
- Reuse of verified software: The deterministic subsystem shows the same behavior on different platforms which satisfy the performance and resource needs of the subsystem, regardless of other differences in each environment, such as existence of unrelated applications. Examples include the different 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.

7.6.1.1 Time Determinism

Each time a calculation is started, its results are guaranteed to be available before a specified deadline. To achieve this, sufficient and guaranteed computing resources (processor time, memory, service response times etc.) should be assigned to the software entities that perform the calculation. For more information on resources see chapter 7.7.

Non-deterministic "best-effort" `Processes` can request guaranteed minimum resources for basic functionality, and additionally can have maximum resources specified for monitoring purposes. However, if Time Determinism is requested, the resources must be guaranteed at any time, i.e. minimum and maximum resources are identical.

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. 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 additionally requires Data Determinism, however in many cases Time Determinism is sufficient.

7.6.1.2 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 always need to be exactly the same, assuming the same initial internal state.

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 dedicated 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.

7.6.1.3 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.

Currently, only Full Deterministic behavior of one Process is supported. Determinism of a cluster of Processes on one or even several machines needs extensions of the Communication Management, which have not been specified yet.

Non-deterministic behavior may arise from different reasons; for example insufficient computing resources, or uncoordinated access of data, potentially by multiple threads running on multiple processor cores. 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 calculations, sporadic unrelated events, race conditions, deviating random numbers etc., i.e. for the same input and initial conditions it always produces the same result within a given time.

7.6.2 Redundant Deterministic Execution

As explained in 7.6.1, future systems need high computing performance in combination with high ASIL safety goals. In this chapter we specify mechanisms which support deterministic multithread execution to support high performance software lockstep solutions. Here are some additional rationales behind it:

- Safety goals for Highly Automated Driving (HAD) systems can be up to ASIL D.
- High Performance Computing (HPC) demands can only be met by non automotive-grade, e.g. consumer electronics (CE), microprocessors, which have high transient hardware error rates compared to automotive-grade microcontrollers. Most likely no such microprocessor is available for ASIL above B, at least for the parts relevant to the design.
- To deal with high error rates, ASIL C/D HAD applications require specific measures, in particular software lockstep, where execution is done redundantly through two different paths and the result is compared to detect errors.
- To make these redundant calculations comparable, software lockstep requires a fully deterministic calculation as defined in 7.6.1.3.
- To meet HPC demands, highly predictable and reliable multi-threading must be supported

Figure 7.13 shows a simplified example for a possible software lockstep architecture.

Two redundant Processes, which run in an internal cycle, get in each cycle the same input data via regular interfaces of Communication Management and produce (in the absence of errors) the same results, due to full deterministic execution.

Execution Management provides DeterministicClient APIs to support control of the process-internal cycle, a deterministic worker pool, activation time stamps and

random numbers. In case of software lockstep, the DeterministicClient interacts with an optional software lockstep framework to ensure identical behavior of the redundantly executed Processes. DeterministicClient interacts with Communication Management to synchronize data handling with cycle activation.

For each execution cycle, the software lockstep framework synchronizes input data in cooperation with Communication Management, makes sure that random numbers and activation time stamps are identical for the redundantly executed Processes, synchronizes triggering of execution, and compares the output to detect failures (e.g. transient processor core or memory errors due to radiation) in one of the redundant Processes. This infrastructure layer can span over multiple hardware instances and is implementation specific.

Details of the software lockstep framework are out of scope of the Adaptive Platform specification. The interaction with DeterministicClient and Communication Management depends on hardware architecture and specific platform design and is a USP of platform providers; so this can only be partly specified in later releases.

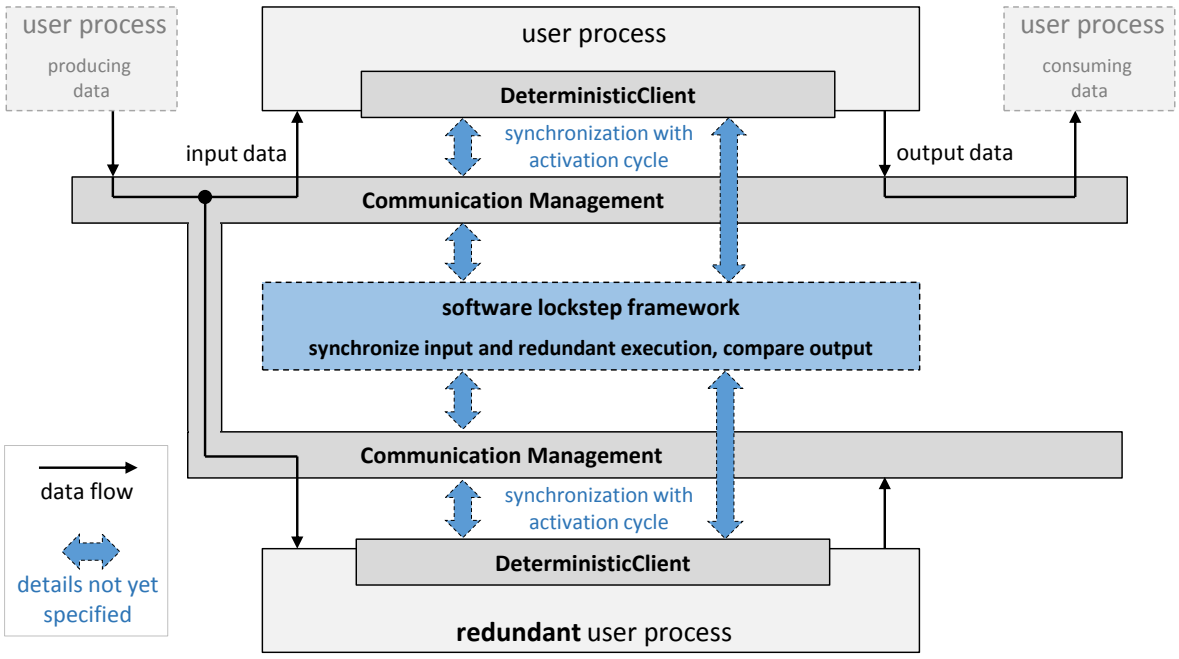

Figure 7.13: Software Lockstep in a typical data flow processing

In case of restart of one of the Processes as an error recovery action due to errors detected when comparing the results, the internal states (i.e. internal memory) need to be resynchronized. To do so, both redundant Processes might need to be re-initialized or even restarted.

Figure 7.14 zooms into one of the redundantly executed Processes.

The AUTOSAR Adaptive Platform needs to provide some library functions to support redundant deterministic execution with sufficient isolation. The library functions (DeterministicClient) run in the context of the user Process.

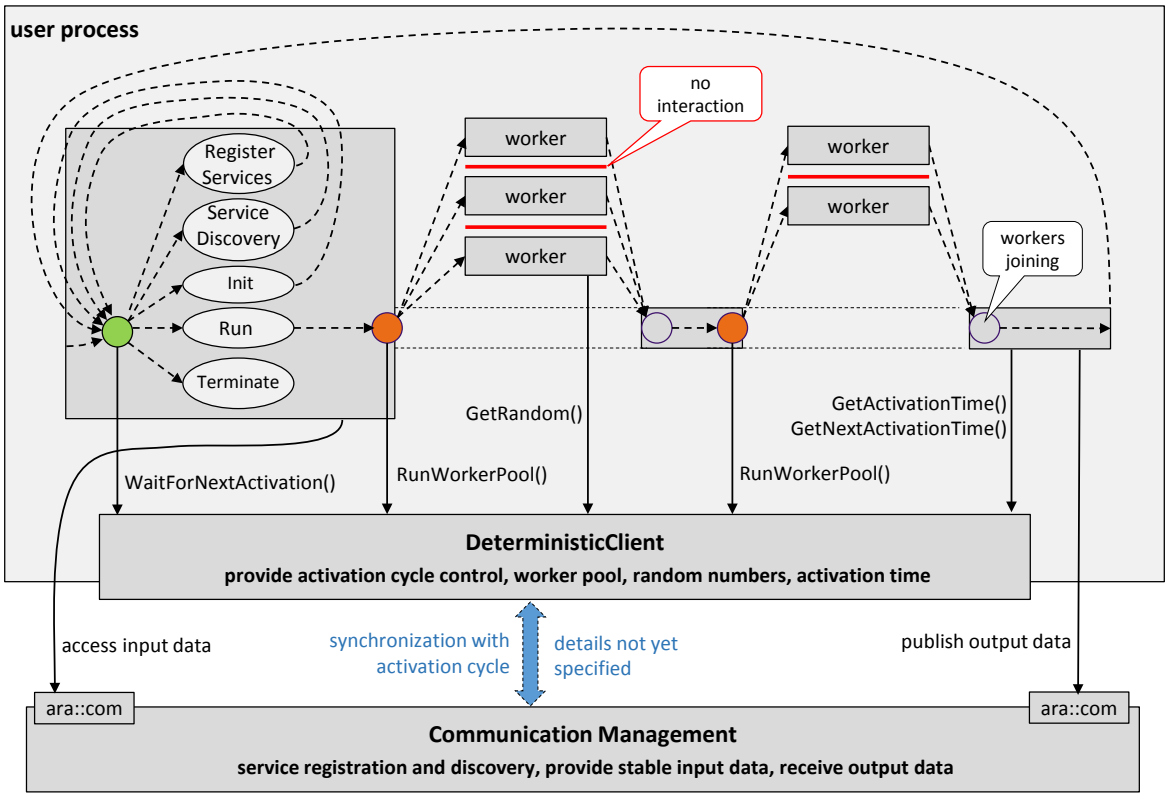

Figure 7.14: Cyclic Deterministic Execution

Cyclic `Process` behavior is controlled by a wait point API. The API returns a code to control the process mode (register services/ service discovery/ init/ run/ terminate). The execution is triggered by the `DeterministicClient`, depending on a defined period or on received events. Within a `Process`, all input data is available via ara::com (polling-based access only) when execution starts and is stable over one execution cycle. For details see 7.6.3.1.

The workload can be deployed to a worker pool API, which allows deterministic execution of a set of container elements (e.g. data sets), which are processed in parallel by the same runnable object (i.e. application function). The runnable object is not allowed to exchange any information while it is running, i.e. it doesn't access data which can be altered by other instances of the runnable object to avoid race conditions. The runnable object instances can physically run in parallel or sequentially in any order. For details see 7.6.3.2.

Additional `DeterministicClient` APIs provide random numbers and activiation time stamps. Common HAD algorithms use particle filters which require random numbers. If used from within the worker pool, the random numbers are assigned to specific container elements to allow deterministic redundant execution. The activation time stamps don't change until the `Process` reaches its next wait point. For deterministic redundant execution, random number seeds and time stamps need to be synchronized. For details see 7.6.3.3 and 7.6.3.4.

At the end of the execution cycle, the `Process` returns to the wait point and waits for the next activation.

The APIs of `DeterministicClient` are standardized and provide abstraction of the application deployment on the actual hardware. The implementation is vendor specific and needs to be configured at integration time individually for each `Process` which uses it.

Different variants of the `DeterministicClient` might work in a software lockstep environment or stand-alone, to support cyclic execution and deterministic worker pools.

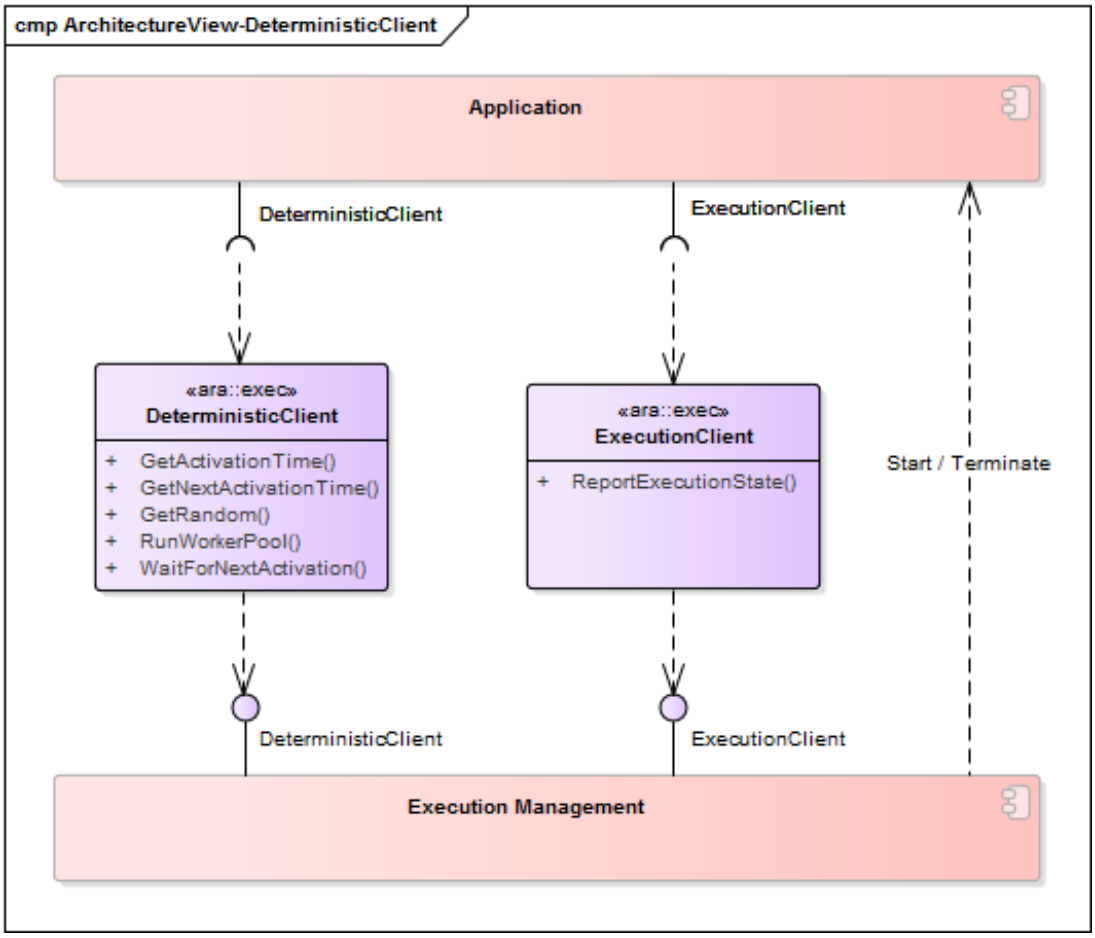

Figure 7.15: Deterministic Execution Interface

7.6.3 Cyclic Deterministic Execution

This section describes the APIs shown in Figure 7.14, and how they need to be used by a `Process` to execute deterministically, so the `Process` can be transparently integrated into a software lockstep environment.

7.6.3.1 Control of Cyclic Execution

Execution Management provides an API to trigger and control recurring, i.e. cyclic execution of the main thread code within a Process.

[SWS_EM_01301]{DRAFT} **Cyclic Execution** ⌈ Execution Management shall provide a blocking wait point API DeterministicClient::WaitForNextActivation. ⌋*(RS_EM_00052, RS_AP_00131)*

After the Process has been started by Execution Management, it reports ExecutionState kRunning (see 7.3.1) and calls DeterministicClient::WaitForNextActivation.

The Process executes one cycle when DeterministicClient::WaitForNextActivation returns and then calls the API again to wait for the next activation.

A return value controls the internal lifecycle (e.g. init, run, terminate) of the Process, see Figure 7.14. The return codes are used to synchronize the behavior of two Processes in case they are executed redundantly.

[SWS_EM_01302]{DRAFT} **Cyclic Execution Control** ⌈ DeterministicClient::WaitForNextActivation shall return a code to control the execution mode of the calling Process. Possible codes are kRegisterServices, kServiceDiscovery, kInit, kRun, and kTerminate. ⌋*(RS_EM_00052, RS_AP_00131)*

The Process returns to DeterministicClient::WaitForNextActivation after each of the following sequential steps:

Register Services – The Process registers its communication services, i.e. the services it offers via Communication Management. This should be the only occasion for performing service registering. No other functionality should be performed in this step to limit resource consumption and runtime, so no dedicated budget needs to be assigned.

Service Discovery – The Process does communication service discovery. This should be the only occasion for performing service discovery, except a service needs to be replaced later (see ([SWS_EM_01304]). No other functionality should be performed in this step to limit resource consumption and runtime, so no dedicated budget needs to be assigned.

Init – The Process initializes its internal data structures. The worker pool (see 7.6.3.2) can be accessed once or several time sequentially. A budget (see 7.6.3.5) needs to be assigned to the “Init” cycle.

Run – The Process performs one cycle of its normal cyclic execution. This can be repeated indefinitely. The worker pool (see 7.6.3.2) can be accessed once or several times sequentially within a cycle. A budget (see 7.6.3.5) needs to be assigned.

Terminate – The Process prepares to terminate. The actual termination is performed according to [SWS_EM_01005], see section 7.3.2.

[SWS_EM_01303]{DRAFT} **Cyclic Execution Control Sequence** ⌈ The return code of DeterministicClient::WaitForNextActivation shall follow this sequence: kRegisterServices, kServiceDiscovery, kInit, kRun, and kTerminate. Only the code kRun can be returned repeatedly, i.e. more than once. ⌋*(RS_EM_00052, RS_AP_00131)*

[SWS_EM_01304]{DRAFT} **Service Modification** ⌈ In case a service which is accessed by the Process needs to be replaced (e.g. due to unavailability) while the kRun cycles are executed, DeterministicClient::WaitForNextActivation shall return kServiceDiscovery once immediately after DeterministicClient::WaitForNextActivation is called, and then continue with the normal kRun cycle. ⌋*(RS_EM_00052, RS_AP_00131)*

The service discovery update needs to be triggered by Communication Management in an implementation specific way. Because the service discovery update runs in addition to the kRun execution within a kRun cycle, the worst case execution time estimation and budget assignment need to consider that kRun and kServiceDiscovery might run sequentially within the configured execution cycle time (see below).

The point in time when DeterministicClient::WaitForNextActivation returns with kRegisterServices, kServiceDiscovery, kInit, kRun (first kRun cycle only, otherwise see below) or kTerminate is implementation specific. In case of redundant execution, the sequences need to be synchronized.

The activation behavior of the kRun-cycles can be realized by Execution Management together with the Communication Management as required by the safety concept. Execution can be triggered via two distinct mechanisms.

- Periodic activation means that DeterministicClient::WaitForNextActivation returns periodically based on a defined period.
- Event-triggered activation means that DeterministicClient::WaitForNextActivation returns based on the communication-event-triggers that are configured for the Process from the outside via Communication Management, e.g. by external units, events generated due to the arrival of data or timer events. Details are out of scope of the Adaptive Platform specification.

[SWS_EM_01351]{DRAFT} **Execution Cycle Time** ⌈ DeterministicClient::WaitForNextActivation shall return with kRun when a configurable cycle time “cycleTimeValue” (see DeterministicClient) has been reached since the last return with kRun (except the kRun-cycle needs to be interrupted or terminated by the implementation specific activation control). ⌋*(RS_EM_00052, RS_AP_00131)*

[SWS_EM_01352]{DRAFT} **Execution Cycle Timeout** ⌈ If the Process calls DeterministicClient::WaitForNextActivation within a kRun cycle after the

configured cycle time “cycleTimeValue” has been exceeded since the last activation, Platform Health Management shall be notified about the timeout to initiate appropriate recovery actions. ⌋(*RS_EM_00052, RS_AP_00131*)

[SWS_EM_01353]{DRAFT} **Event-triggered Cycle Activation** ⌈ If the configured cycle time “cycleTimeValue” is zero, DeterministicClient::WaitForNextActivation shall be triggered by Communication Management to start the next kRun cycle. The trigger conditions are implementation specific and evaluated by Communication Management. ⌋(*RS_EM_00052, RS_AP_00131*)

This cyclic behavior can be used in a software lockstep environment to initialize and trigger execution of redundant Processes and compare the results after a cycle has finished. For redundant execution, the execution behavior and its budget (activation timing, computing time, computing resources) should be explicitly visible at integration time to configure Execution Management accordingly.

Execution Management together with Communication Management initiates service discovery so that in total the behavior is deterministic. Optionally, e.g. if necessary for a software lockstep implementation, all input data as received via Communication Management should be available when a cycle starts and guaranteed to be deterministically consistent.

7.6.3.2 Worker Pool

[SWS_EM_01305]{DRAFT} **Worker Pool** ⌈ Execution Management shall provide a blocking API DeterministicClient::RunWorkerPool to run a deterministic worker pool to be used within the Process execution cycle. ⌋(*RS_EM_00053, RS_AP_00131*)

The worker pool is triggered by the main-thread of the Process in a sequential order. DeterministicClient::RunWorkerPool is blocking and therefore there is no parallelism between the main-thread and the worker pool. The user Process is not allowed to create threads on its own by using normal POSIX mechanisms to avoid the risk of inducing indeterministic behavior.

DeterministicClient::RunWorkerPool registers a “worker” runnable object, along with its parameter object. The parameter contains a set of objects, which are processed in parallel by the same runnable object invoked from multiple workers (e.g. based on POSIX threads) in the pool (see Figure 7.16). This means, the deterministic worker pool is used to process a set of container elements, which are the parameters to the worker. Each element in the container represents a job to be computed. The deterministic distribution of the elements to individual workers is done by using the container iterator.

[SWS_EM_01306]{DRAFT} **Processing Container Objects** ⌈ DeterministicClient::RunWorkerPool shall sequentially (using the iterator of input parameter “container”) call a method workerRunnable(...) (input parameter “runnableObj”) on

every element of “container”, by using a worker pool of size “numberOfWorkers”. ⌋ *(RS_EM_00053, RS_AP_00131)*

The implementation and size of the worker pool (i.e. number of threads) is hidden from the user. The Integrator decides about the size and the implementation and configures a parameter “numberOfWorkers” (see `DeterministicClient`). The distribution of the worker threads to processor cores is left to the Operating System.

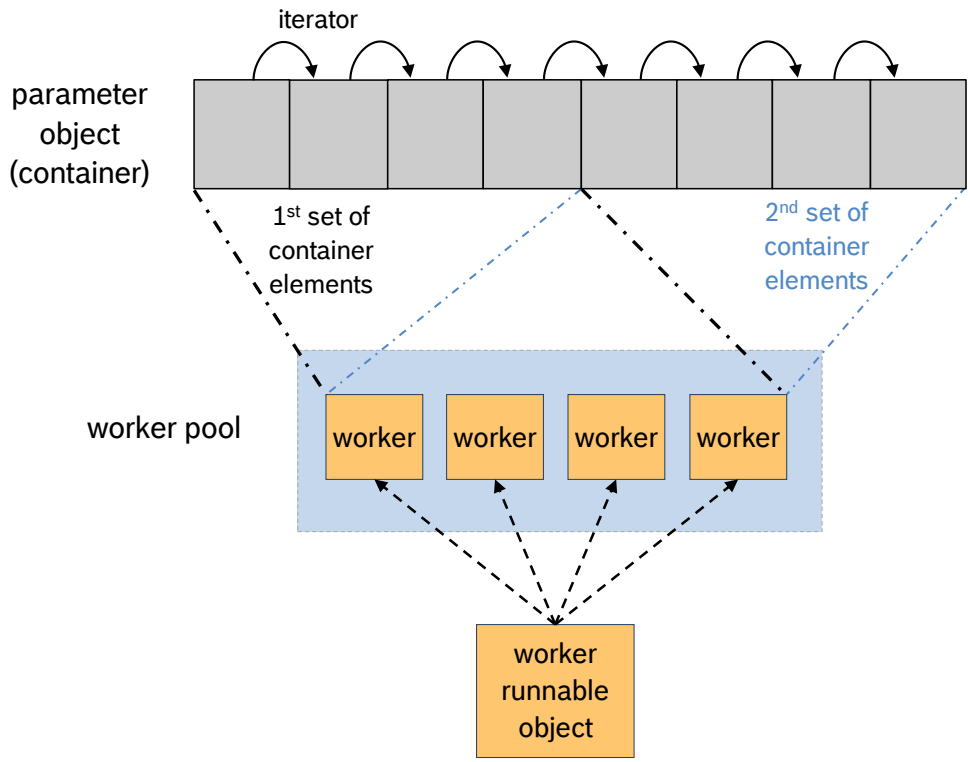

Figure 7.16: Worker Pool Usage

If the number of required container elements exceeds the number of workers (threads) in the deterministic worker pool, `Execution Management` can use the worker pool several times sequentially (with unrestricted interleaving), which shall be transparent to the user of the worker pool.

To achieve Data Determinism, the parallel workers need to satisfy certain implementation properties, e.g. no exchange of data is allowed between the instances of the runnable object which are processed by the workers. For details see [11]. Other, more complex solutions which allow interaction between the workers would be possible, but they increase complexity, reduce utilization and transparency, and are error-prone regarding the deterministic behavior.

The worker pool runs within the `Process` context of the caller of this API. It is designed as part of `Execution Management` to guarantee the deterministic behavior by incorporating it in the `DeterministicClient::WaitForNextActivation` cycle.

An example for the implementation of a worker runnable object can be found in [11].

The aim is to abstract the data processing as far as possible, irrespective of the actual number of available parallel execution paths. Example: a task with N similar subtasks (e.g. N Kalman-filters). The task is assigned to the worker pool and the worker pool processes it using a given worker runnable object (in this example the worker runnable object would be the Kalman-filter).

The worker pool cannot be used to process multiple different tasks in parallel. The use of multiple potentially different explicit functions (worker runnable objects) could add unnecessary complexity and can lead to extremely heterogeneous runtime utilization, as each worker may have different computing time. This would complicate the planning of resource deployment, which is necessary for black-box integration.

7.6.3.3 Random Numbers

[SWS_EM_01308]{DRAFT} **Random Numbers** ⌈ `Execution Management` shall provide an API `DeterministicClient::GetRandom` which provides "Deterministic" random numbers. 'Deterministic" means, that the provided random numbers are identical for `Processes` which are executed redundantly, including within runnable objects being processed by a worker pool (see [SWS_EM_01305]). ⌋*(RS_EM_00053, RS_AP_00131)*

If used from within `DeterministicClient::RunWorkerPool`, the random numbers are assigned to specific container elements, using the container iterator, to allow deterministic redundant execution.

The provision of the seeds for the pseudo random numbers generation is designed as part of `Execution Management` to guarantee the deterministic behavior by incorporating it in the `DeterministicClient::WaitForNextActivation` cycle.

Implementations of `DeterministicClient` which do not need to support redundant execution can provide standard random numbers without specific properties.

7.6.3.4 Time Stamps

The deterministic user `Process` might need timing information while cyclically (see 7.6.3.1) processing its input data in the kRun cycle. The used time value may have an influence on the calculated results. Therefore, `Execution Management` returns deterministic timestamps that represent the points in time when the current cycle was activated and when the next cycle will be activated, if this value is known. The timestamps are required to be identical for `Processes` which are executed redundantly, e.g. in a lockstep environment (see 7.6.2).

[SWS_EM_01310]{DRAFT} **Get Activation Time** ⌈ `Execution Management` shall provide an API `DeterministicClient::GetActivationTime` which provides a deterministic timestamp that represents the point in time when the current kRun

cycle was activated by DeterministicClient::WaitForNextActivation (see [SWS_EM_01301]). Deterministic means, that the timestamps are identical for Processes which are executed redundantly. Subsequent calls within a cycle shall always return the same value. ⌋*(RS_EM_00053, RS_AP_00131)*

[SWS_EM_01311]{DRAFT} **Activation Time Unknown** ⌈ If DeterministicClient::GetActivationTime is called from outside a kRun cycle, Execution Management shall return kNotAvailable. ⌋*(RS_EM_00053, RS_AP_00131)*

[SWS_EM_01312]{DRAFT} **Get Next Activation Time** ⌈ Execution Management shall provide an API DeterministicClient::GetNextActivationTime which provides a deterministic timestamp that represents the point in time when the next kRun cycle will be activated by DeterministicClient::WaitForNextActivation (see [SWS_EM_01301]). Deterministic means, that the timestamps are identical for Processes which are executed redundantly. Subsequent calls within a cycle shall always return the same value. ⌋*(RS_EM_00053, RS_AP_00131)*

[SWS_EM_01313]{DRAFT} **Next Activation Time Unknown** ⌈ In case the next activation time is not known when calling DeterministicClient::GetNextActivationTime, e.g. because of non-equidistant event-triggered activation, Execution Management shall return kNotAvailable. ⌋ *(RS_EM_00053, RS_AP_00131)*

7.6.3.5 Real-Time Resources

To ensure Time Determinism (see 7.6.1.1), i.e. to make sure that a cyclic deterministic execution within a Process (see 7.6.3.1) is finished at a given deadline we need:

- Execution Management supports deterministic multithreading to meet high performance demand, see 7.6.3.2
- The integrator needs to assign appropriate resources to the Process.
- The integrator needs to assign appropriate scheduling policies. Details and options other than standard POSIX scheduling policies (see [SWS_EM_01014]) heavily depend on the used Operating System, are vendor specific, and are for now out of scope of the Adaptive Platform specification.
- The integrator needs to configure deadline monitoring, possibly execution budget monitoring, and appropriate recovery actions in case of violations. For more details on resources see 7.7.

To make sure that all Processes which use the DeterministicClient APIs get enough computing resources and can finish their cycle in time, it is in particular important to know when the worker pool (DeterministicClient::RunWorkerPool) is needed within a kInit and kRun DeterministicClient::WaitForNextActivation cycle. Also, a good computing resource utilization can only be achieved if usage of the workers (i.e. of available cores) can be distributed evenly over time. If the application code is known to the

integrator, it should not be a problem to analyze the behavior and configure the system accordingly. However, if third party “black box” applications are delivered for integration, their resource demands need to be described in a standardized way, so the integrator has a rough idea about the distribution of resource consumption within a `DeterministicClient::WaitForNextActivation`-cycle.

To describe budget needs within the kInit and kRun cycle, we use a normalized value `NormalizedInstruction` to specifiy runtime consumption on the target system.

`NormalizedInstruction` = runtime in sec * clock frequency in Hz

`NormalizedInstruction` does not reflext the actual number of code instructions, but allows the description of comparative resource needs.

The following parameters (`DeterministicClientResource`, see [TPS_MANI_01200] in [4]) are relevant for describing the computing time budget needs of a `Process` which uses `DeterministicClient::RunWorkerPool`.

The parameters are needed to be specified twice per `Process` which uses `DeterministicClient`, once for the kInit cycle and once for the kRun cycles (`DeterministicClientResourceNeeds`, and [TPS_MANI_01199]).

- *numberOfInstructions* [NormalizedInstructions]

  This is the normalized runtime consumption on the target system within one cycle, assuming the “worst-case” runtime where the workers would be executed sequentially.

- *speedup* = sequental runtime / parallelized runtime

  Defines how much faster the calculations within one cycle can be finished if *numberOfWorkers* (see 7.6.3.2) are physically available, i.e. if enough cores were available on the machine to perform parallel execution of all workers.

- *sequentialInstructionsBegin* [NormalizedInstructions]

  This is the normalized sequential runtime at the beginning of the cycle (which mostly cannot be parallelized), before the main usage of the worker pool starts.

- *sequentialInstructionsEnd* [NormalizedInstructions]

  This is the normalized sequential runtime at the end of the cycle (which mostly cannot be parallelized), after the main usage of the worker pool has ended.

Examples

Example 7.4

The `Process` uses the worker pool mainly in the middle of the cycle. The first 100 (normalized) instructions are mostly sequential, the next 275 instructions have a benefit

when using the worker pool, and the last 125 instructions are mostly sequential again. The average speedup, over the complete 500 instructions is 1.3.

- *numberOfInstructions* = 500
- *numberOfWorkers* = 2
- *speedup* = 1.3
- *sequentialInstructionsBegin* = 100
- *sequentialInstructionsEnd* = 125

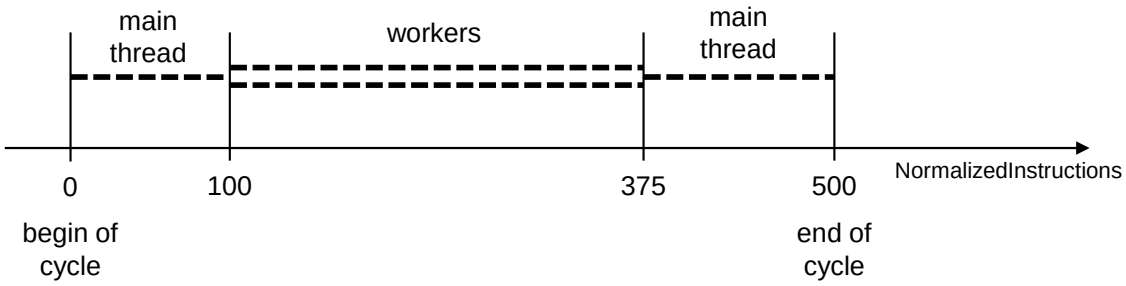

Figure 7.17: Worker pool used in middle of cycle

Example 7.5

The `Process` runs sequentially throughout most of the cycle and does not benefit in using the worker pool, i.e. the overhead of using the worker pool compensates the parallelization gain.

- *numberOfInstructions* = 200
- *numberOfWorkers* = 2
- *speedup* = 1
- *sequentialInstructionsBegin* = 200
- *sequentialInstructionsEnd* = 0

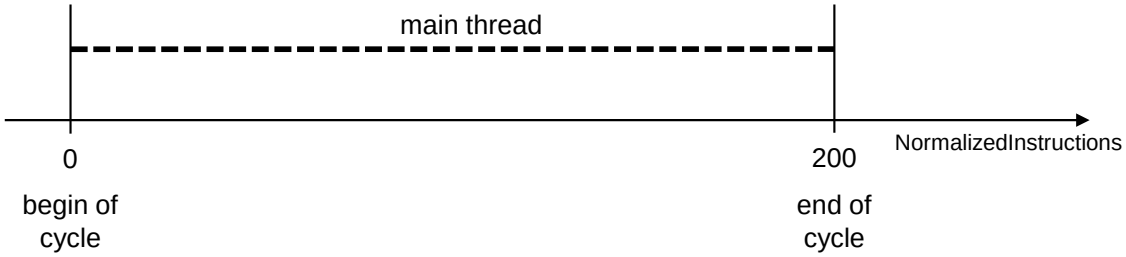

Figure 7.18: No benefit from worker pool

Example 7.6

The `Process` fully utilizes the worker pool throughout the cycle.

- *numberOfInstructions* = 200
- *numberOfWorkers* = 3
- *speedup* = 2.9
- *sequentialInstructionsBegin* = 0
- *sequentialInstructionsEnd* = 0

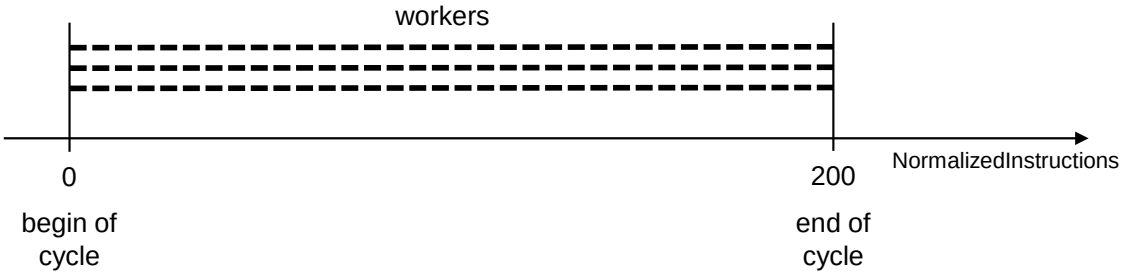

Figure 7.19: Full utilization of worker pool

7.7 Resource Limitation

Despite the correct behavior of a particular Adaptive Application in the system, it is important to ensure any potentially incorrect behavior, as well as any unforeseen interactions cannot cause interference in unrelated parts of the system [RS_EM_00002]. As AUTOSAR Adaptive Platform also strives to allow consolidation of several functions on the same machine, ensuring Freedom From Interference is a key property to maintain.

However, AUTOSAR 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.

In addition, it is important to consider that Execution Management is only responsible for the correct configuration of the Machine. However, enforcing the associated restrictions is usually done by either the Operating System or another Application like the Persistency service.

Some mechanisms that could be standardized will not yet be defined in this release.

7.7.1 Resource Configuration

This section provides an overview on resource assignment to Processes. The resources considered in this specification are:

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

Other resources like persistent storage or I/O usage are also relevant, but are currently out of scope for this specification.

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.

The following stakeholders are involved in resource management:

- Application Developer

  The Application developer should know how much memory (RAM) and computing resources the Processes 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 Execution 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.

Therefore, while the Application developer should be able to bring estimates regarding the resource consumption, a precise usage cannot be provided out of context.

- 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 should assign available resources to the applications which can be active at the same time, which is closely related to `State Management` configuration, see section 7.4. 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.
  - 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 7.7.3.1.
  - 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 should 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 7.7.3.4.

Based on the above, all the resource configuration elements are to be configured during platform integration, most probably by the Integrator. To group these configuration elements, we define a `ResourceGroup`. It may have several properties configured to enable restricting `Applications` running in the group. Subsequently, each `Process`

is required to belong to a ResourceGroup, clarifying how the Application will be constrained at the system level.

[SWS_EM_02102]{DRAFT} **Memory control** ⌈ Execution Management shall configure the maximum amount of RAM available globally for all Processes belonging to each ResourceGroup when defined in the configuration, before loading a Process from this ResourceGroup. ⌋*(RS_EM_00005, RS_AP_00131)*

If a ResourceGroup does not have a configured RAM limit, then the Processes are only bound by their implicit memory limit.

[SWS_EM_02103]{DRAFT} **CPU usage control** ⌈ Execution Management shall configure the maximum amount of CPU time available globally for all Processes belonging to each ResourceGroup when defined in the configuration, before loading a Process from this ResourceGroup. ⌋*(RS_EM_00005, RS_AP_00131)*

If ResourceGroup does not have a configured CPU usage limit, then the Processes are only bound by their implicit CPU usage limit (priority, scheduling scheme...).

7.7.2 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 7.7.3.1. For RAM monitoring see 7.7.3.4. 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) which are provided by Platform Health Management.

Resource monitoring can serve several purposes, e.g.

- Detection of misbehavior 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), 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 7.6.1).

- 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 AUTOSAR Adaptive Platform.

7.7.3 Application-level Resource configuration

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.

7.7.3.1 CPU Usage

CPU usage is represented in a Process by its threads. Generally speaking, Operating Systems use some properties of each thread's configuration to determine when to run it, and additionally constrain a group of threads to not use more than a defined amount of CPU time. Because threads may be created at runtime, only the first thread can be configured by Execution Management.

7.7.3.2 Core Affinity

[SWS_EM_02104]{DRAFT} **Core affinity** ⌈ Execution Management shall configure the Core affinity of the Process initial thread restricting it to a sub-set of cores in the system. ⌋*(RS_EM_00008, RS_AP_00131)*

Requirement [SWS_EM_02104] permits the initial thread (the "main" thread of the Process) to be bound to certain cores [SWS_OSI_01012]. Depending on the capabilities of the Operating System the sub-set could be a single core. If the Operating System does not support binding to specific cores then the only supported sub-set is the entire set of cores.

7.7.3.3 Scheduling

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 AUTOSAR Adaptive Platform to enforce a particular scheduling strategy for an entire Process, but only for its first thread.

[SWS_EM_01014]{DRAFT} **Scheduling policy** ⌈ Execution Management shall support the configuration of the scheduling policy when launching a Process, based on information provided by the Execution Manifest. ⌋*(RS_EM_00002, RS_AP_00131)*

For the detailed definitions of these policies, refer to [12]. Note, `SCHED_OTHER` shall be treated as non real-time scheduling policy, and actual behavior of the policy is implementation specific. It should not be assumed that the scheduling behavior is compatible between different AUTOSAR Adaptive Platform implementations, except that it is a non real-time scheduling policy in a given implementation.

- **[SWS_EM_01041]**{DRAFT} **Scheduling FIFO** ⌈ Execution Management shall be able to configure FIFO scheduling using policy `SCHED_FIFO`. ⌋ *(RS_EM_00002, RS_AP_00131)*
- **[SWS_EM_01042]**{DRAFT} **Scheduling Round-Robin** ⌈ Execution Management shall be able to configure round-robin scheduling using policy `SCHED_RR`. ⌋*(RS_EM_00002, RS_AP_00131)*
- **[SWS_EM_01043]**{DRAFT} **Scheduling Other** ⌈ Execution Management shall be able to configure non real-time scheduling using policy `SCHED_OTHER`. ⌋ *(RS_EM_00002, RS_AP_00131)*

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

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

[SWS_EM_01015]{DRAFT} **Scheduling priority** ⌈ Execution Management shall support the configuration of a scheduling priority when launching a Process, based on information provided by the Execution Manifest. ⌋*(RS_EM_00002, RS_AP_00131)*

The available priority range and actual meaning of the scheduling priority depends on the selected scheduling policy, see [constr_1620] and [constr_1621] in [4].

7.7.3.3.1 Resource Management

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.

The remaining critical property however is to guarantee that a particular process or set of Processes cannot adversely affect the behavior of other Processes.

To guarantee Full Determinism for the entire system, it is important to ensure Freedom from Interference, which the ResourceGroup contribute to ensure.

[SWS_EM_02106]{DRAFT} **ResourceGroup assignment** ⌈ Execution Management shall configure the Process according to its ResourceGroup membership. ⌋ *(RS_EM_00005, RS_AP_00131)*

7.7.3.4 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 should 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.

Execution Management should ensure that the entire memory range that deterministic Processes may be using is available at the start and for the whole duration of the respective Process 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 a Process, some properties may need to be defined for each Process.

[SWS_EM_02107]{DRAFT} **Maximum heap** ⌈ Execution Management shall configure the Maximum heap usage for the Process. ⌋*(RS_EM_00005, RS_AP_00131)*

Heap memory is used for dynamic memory allocation inside a Process e.g. through malloc()/free() and new/delete.

[SWS_EM_02108]{DRAFT} **Maximum system memory usage** ⌈ Execution Management shall configure the Maximum system memory usage of the Process. ⌋ *(RS_EM_00005, RS_AP_00131)*

System memory can be used to create extra resources like file handles or semaphores, as well as creating new threads.

[SWS_EM_02109]{DRAFT} **Process pre-mapping** ⌈ Execution Management shall pre-map a Process if required by the corresponding Execution Manifest. ⌋*(RS_EM_00005, RS_AP_00131)*

Fully pre-mapping a Process ensures that code and data execution is not going to be delayed at its first execution by demand-loading. This helps providing Time Determinism during system startup and first execution phases, but also helps with safety where code handling error cases can be preloaded and made guaranteed to be available. In addition, pre-mapping avoids late issues where filesystem may be corrupted and part of the Process may not be loadable anymore.

7.8 Fault Tolerance

7.8.1 Introduction

What is Fault-Tolerance?

The method of coping with faults within a large-scale software system is termed fault tolerance.

The model adopted for Execution Management is outlined in [13].

This section provides context to the application of fault tolerance concepts with respect to Execution Management and perspective on how this contributes in overall platform instance's dependability.

Platform-wide Service Oriented Architecture fault tolerance aspects are outside the scope of this document and are not further addressed.

7.8.2 Scope

Execution Management has a crucial influence on overall system behavior of the AUTOSAR Adaptive Platform.

The effect of erroneous functionality, within Execution Management can have very different severity depending on operational mode and fault type. For example, a fault identified by Execution Management may have a local effect, influencing an independent process only, or may become a root cause for a Machine wide failures.

It is therefore necessary to not only specify correct behavior but also to introduce alternative behavior in case of deviations.

Such mechanisms address a broad spectrum of concerns that emerge during Machine and Process Life Cycle Management.

The AUTOSAR Adaptive Platform architecture is composed of two levels; Application and Platform Instance. The Application level constitutes cooperative Applications intended to satisfy overal system's needs and objectives and represents a service level in vehicle context. The Platform Instance level as a reusable asset providing basic capabilities and platform level services. Fault tolerance within Execution Management is therefore required to handle both levels.

7.8.3 Threat Model

The main threats which leading to incorrect behavior of software - whether Application or Platform Instance - is the presence of systematic defects or faults i.e. those incorporated during design phase and remaining dormant untill deployment. Other sources of faults include physical faults, e.g. random hardware failures, that

might influence resource allocation and correct execution, and interraction faults which can be a source for incorrect state transition requests.

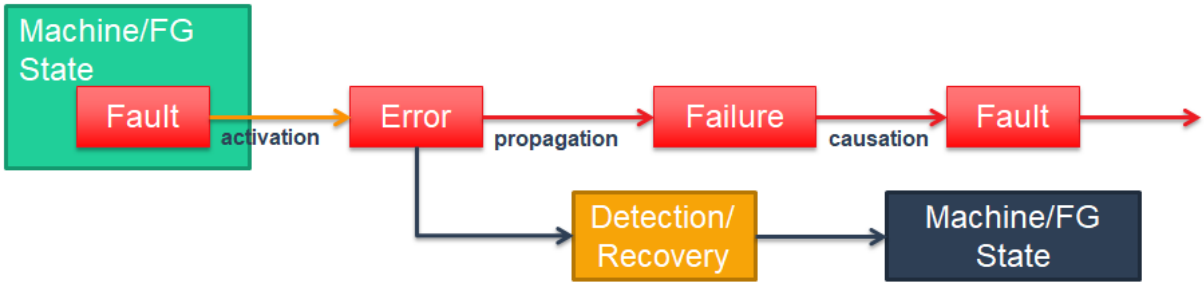

Figure 7.20: General Fault Tolerance scheme.

From the perspective of `Execution Management`, fault activation occures when resulting `Function Group State` or combination of such is requested. Due to the different nature of faults, these can lead to various types of deviations from expected functional behavior and finally result in erroneous system functionality either in terms of correct computational results or timing response.

In general, the implementation of fault tolerance mechanism is based on two consistent steps - `Error Detection` and subsequent `Error Recovery`. The major focus of `Error Detection` during `Design Phase` activities and thus the focus of `Fault Tolerance` in this specification is on the analysis of potential `Failure Modes` and the consequent error detection mechanisms that should later be incorporated into the implementation.

In contrast, `Error Recovery` consists of actions that should be taken in order to restore the system's state where the system can once again perform correct service delivery. Binding of `Error Detection` and `Recovery Actions` should be a subject of platform wide fault tolerance model.

Remark: The remainder of this section is the subject for elaboration for the next release of this specification. Provision for fault-tolerance mechanisms will consider possible faults, how they can lead to errors within `Execution Management` and the mechanisms that are introduced to ensure error detection.

8 API specification

8.1 Type Definitions

8.1.1 ExecutionState

[SWS_EM_02000]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::exec::ExecutionState	
Scope:	namespace ara::exec	
Values:	kRunning= 0	–
	kTerminating= 1	–
Header file:	#include "ara/exec/execution_client.h"	
Description:	Defines the internal states of a Process (see 7.3.1). Scoped Enumeration of uint8_t .	

⌋(*RS_EM_00103*)

Please note that ExecutionState includes only states reportable by the `Process` to `Execution Management` and therefore does not include enumerations e.g. the "Initializing" state mentioned in figure 7.3 and 7.9, which are an implied states for `Execution Management`. The Initializing state starts when `Process` is first scheduled (so no code executed yet) and ends when `kRunning` is reported. For the reasons mentioned, `Execution Management` assumes that `Process` is in initializing state until `kRunning` will be reported by it.

8.1.2 ExecutionReturnType

[SWS_EM_02070]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::exec::ExecutionStateReturnType	
Scope:	namespace ara::exec	
Values:	kSuccess= 0	–
	kGeneralError= 1	–
Header file:	#include "ara/exec/execution_client.h"	
Description:	Defines the error codes for ExecutionClient operations. Scoped Enumeration of uint8_t .	

⌋(*RS_EM_00101*)

8.1.3 ActivationReturnType

[SWS_EM_02201]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::exec::ActivationReturnType	
Scope:	namespace ara::exec	
Values:	kRegisterServices= 0	application shall register communication services(this must be the only occasion for performing service registering)
	kServiceDiscovery= 1	application shall do communication service discovery (this must be the only occasion for performing service discovery)
	kInit= 2	application shall initialize its internal data structures (once)
	kRun= 3	application shall perform its normal operation
	kTerminate= 4	application shall terminate
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	Defines the return codes for WaitForNextActivation operations. Scoped Enumeration of uint8_t .	

⌋(*RS_EM_00052*)

8.1.4 ActivationTimeStampReturnType

[SWS_EM_02202]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::exec::ActivationTimeStampReturnType	
Scope:	namespace ara::exec	
Values:	kSuccess= 0	–
	kNotAvailable= 1	–
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	Defines the return codes for "get activation timestamp" operations. Scoped Enumeration of uint8_t .	

⌋(*RS_EM_00053*)

8.2 Class Definitions

8.2.1 ExecutionClient class

The Execution State API provides the functionality for a `Process` to report its state to the `Execution Management`.

[SWS_EM_02001]{DRAFT} ⌈

Kind:	class
Symbol:	ara::exec::ExecutionClient
Scope:	namespace ara::exec
Syntax:	`class ExecutionClient {...};`
Header file:	#include "ara/exec/execution_client.h"
Description:	Class to implement operations on Execution Client. .

⌋(*RS_EM_00103*)

8.2.1.1 ExecutionClient::ExecutionClient

[SWS_EM_02030]{DRAFT} ⌈

Kind:	function
Symbol:	ara::exec::ExecutionClient::ExecutionClient()
Scope:	class ara::exec::ExecutionClient
Syntax:	`ExecutionClient () const noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/exec/execution_client.h"
Description:	Constructor that creates the Execution Client. .
Notes:	Constructor for ExecutionClient which opens the Execution Management communication channel (e.g. POSIX FIFO) for reporting the Execution State. Each Process shall create an instance of this class to report its state

⌋(*RS_EM_00103*)

8.2.1.2 ExecutionClient::~ExecutionClient

[SWS_EM_02002]{DRAFT} ⌈

Kind:	function
Symbol:	ara::exec::ExecutionClient::~ExecutionClient()
Scope:	class ara::exec::ExecutionClient
Syntax:	`~ExecutionClient () const noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/exec/execution_client.h"
Description:	Destructor of the Execution Client instance. .

⌋(*RS_EM_00103*)

8.2.1.3 ExecutionClient::ReportExecutionState

[SWS_EM_02003]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::ExecutionClient::ReportExecutionState(ExecutionState state)	
Scope:	class ara::exec::ExecutionClient	
Syntax:	`ExecutionReturnType ReportExecutionState (ExecutionState state) const noexcept;`	
Parameters (in):	state	Value of the Execution State
Return value:	ExecutionReturnType	–
Exception Safety:	noexcept	
Header file:	#include "ara/exec/execution_client.h"	
Description:	Interface for a Process to report its internal state to Execution Management. .	

⌋(*RS_EM_00103*)

8.2.2 DeterministicClient class

The `DeterministicClient` class provides the functionality for an `Application` to run a cyclic deterministic execution, see 7.6.3. Each `Process` which needs support for cyclic deterministic execution has to instantiate this class.

[SWS_EM_02210]{DRAFT} ⌈

Kind:	class
Symbol:	ara::exec::DeterministicClient
Scope:	namespace ara::exec
Syntax:	`class DeterministicClient {...};`
Header file:	#include "ara/exec/deterministic_client.h"
Description:	Class to implement operations on Deterministic Client .

⌋(*RS_EM_00052*)

8.2.2.1 DeterministicClient::DeterministicClient

[SWS_EM_02211]{DRAFT} ⌈

Kind:	function
Symbol:	ara::exec::DeterministicClient::DeterministicClient()
Scope:	class ara::exec::DeterministicClient
Syntax:	`DeterministicClient () const noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/exec/deterministic_client.h"

▽

△

Description:	Constructor for DeterministicClient which opens the Execution Management communication channel (e.g. POSIX FIFO) to access a wait point for cyclic execution, a worker pool, deterministic random numbers and time stamps .

⌋(*RS_EM_00052*, *RS_EM_00053*)

8.2.2.2 DeterministicClient::~DeterministicClient

[SWS_EM_02215]{DRAFT} ⌈

Kind:	function
Symbol:	ara::exec::DeterministicClient::~DeterministicClient()
Scope:	class ara::exec::DeterministicClient
Syntax:	`~DeterministicClient () const noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/exec/deterministic_client.h"
Description:	Destructor of the Deterministic Client instance .

⌋(*RS_EM_00052*, *RS_EM_00053*)

8.2.2.3 DeterministicClient::WaitForNextActivation

[SWS_EM_02216]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::DeterministicClient::WaitForNextActivation()	
Scope:	class ara::exec::DeterministicClient	
Syntax:	`ActivationReturnType WaitForNextActivation () const noexcept;`	
Return value:	ActivationReturnType	–
Exception Safety:	noexcept	
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	Blocks and returns with a process control value when the next activation is triggered by the Runtime .	

⌋(*RS_EM_00052*)

8.2.2.4 DeterministicClient::RunWorkerPool

[SWS_EM_02220]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::DeterministicClient::RunWorkerPool(Worker &runnableObj, Container &container)	
Scope:	class ara::exec::DeterministicClient	
Syntax:	`Void RunWorkerPool (Worker &runnableObj, Container &container) const noexcept;`	
Parameters (in):	runnableObj	Object that provides a method called worker-Runnable (...), which will be called on every container element
	container	C++ container which supports a standard iterator interface with - begin() - end() - operator*() operator++
Return value:	Void	–
Exception Safety:	noexcept	
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	Uses a worker pool to call a method Worker::workerRunnable (...) for every element of the container. The sequential iteration is guaranteed by using the container++ operator. The API guarantees that no other iteration scheme is used .	

⌋(*RS_EM_00053*)

8.2.2.5 DeterministicClient::GetRandom

[SWS_EM_02225]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::DeterministicClient::GetRandom()	
Scope:	class ara::exec::DeterministicClient	
Syntax:	`uint64_t GetRandom () const noexcept;`	
Return value:	uint64_t	uint64_t 64 bit uniform distributed pseudo random number
Exception Safety:	noexcept	
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	This returns “Deterministic” random numbers. ‘Deterministic” means, that the returned random numbers are identical within redundant DeterministicClient::WaitForNextActivation() cycles, which are used within redundantly executed Process .	

⌋(*RS_EM_00053*)

8.2.2.6 DeterministicClient::GetActivationTime

[SWS_EM_02230]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::DeterministicClient::GetActivationTime(TimeStamp)	
Scope:	class ara::exec::DeterministicClient	
Syntax:	`ActivationTimeStampReturnType GetActivationTime (TimeStamp) const noexcept;`	
DIRECTION NOT DEFINED	TimeStamp	–
Return value:	ActivationTimeStampReturnType	–
Exception Safety:	noexcept	
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	This provides the timestamp that represents the point in time when the activation was triggered by ::WaitForNextActivation() with return value kRun. Subsequent calls within an activation cycle will always provide the same value. The same value will also be provided within redundantly executed Processes .	

⌋(*RS_EM_00053*)

8.2.2.7 DeterministicClient::GetNextActivationTime

[SWS_EM_02235]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::exec::DeterministicClient::GetNextActivationTime(TimeStamp)	
Scope:	class ara::exec::DeterministicClient	
Syntax:	`ActivationTimeStampReturnType GetNextActivationTime (TimeStamp) const noexcept;`	
DIRECTION NOT DEFINED	TimeStamp	–
Return value:	ActivationTimeStampReturnType	–
Exception Safety:	noexcept	
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	This provides the timestamp that represents the point in time when the next activation will be triggered by {DeterministicClient::WaitForNextActivation}() with return value kRun. Subsequent calls within an activation cycle will always provide the same value. The same value will also be provided within redundantly executed {Process} .	

⌋(*RS_EM_00053*)

9 Service Interfaces

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

There are no service interfaces defined in this release.

A Mentioned Manifest Elements

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::ExecutionManifest
Note	This enum defines the different styles how the command line option appears in the command line. **Tags:** atp.Status=draft
Literal	***Description***
commandLineLong Form	Long form of command line option. Example: –version=1.0 –help **Tags:** atp.EnumerationValue=1
commandLineShort Form	Short form of command line option. Example: -v 1.0 -h **Tags:** atp.EnumerationValue=0
commandLine SimpleForm	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 A.1: CommandLineOptionKindEnum

Class	**DeterministicClient**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	The meta-class DeterministicClient provides the ability to support the deterministic execution of one or more processes with specific configuration parameters for DeterministicClient library functions. **Tags:** atp.Status=draft atp.recommendedPackage=DeterministicClients			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadablePackageElement*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
cycleTimeValue	TimeValue	0..1	attr	This attribute represents the cycle time for execution of a DeterministicClient activation cycle.
numberOf Workers	PositiveInteger	0..1	attr	Number of independent workers that process data-sets. Size of the worker pool shall be decided based on availability of resources like processor cores or memory.

Table A.2: DeterministicClient

Class	**DeterministicClientResource**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ProcessDesign			
Note	This meta-class specifies computing resource needs of DeterministicClient library functions. **Tags:** atp.Status=draft			
Base	*ARObject*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
numberOf Instructions	NormalizedInstruction	0..1	attr	This attribute represents the normalized runtime consumption on the target system within one DeterministicClient::WaitForNextActivation cycle, assuming the "worst-case" runtime where the workers would be executed sequentially.
sequential Instructions Begin	NormalizedInstruction	0..1	attr	Normalized sequential runtime at the beginning of the DeterministicClient::WaitForNextActivation cycle (which mostly cannot be parallelized), before the main usage of the worker pool starts.
sequential InstructionsEnd	NormalizedInstruction	0..1	attr	WaitForNextActivation cycle (which mostly cannot be parallelized), after the main usage of the worker pool has ended.
speedup	Float	0..1	attr	This attribute defines how much faster the calculations within one DeterministicClient::WaitForNextActivation cycle can be finished if numberOfWorkers are physically available, i.e. if enough cores were available on the machine to perform parallel execution of all workers (sequential runtime / parallelized runtime).

Table A.3: DeterministicClientResource

Class	**DeterministicClientResourceNeeds**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ProcessDesign			
Note	This meta-class specifies process and cycle specific computing resource needs of DeterministicClient library functions. **Tags:** atp.Status=draft			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
hardware Platform	String	0..1	attr	This attribute represents a textual identification of the target platform.
initResource	DeterministicClient Resource	0..1	aggr	This represents the computing resource needs of a DeterministicClient::WaitForNextActivation kInit cycle. **Tags:** atp.Status=draft
runResource	DeterministicClient Resource	0..1	aggr	This represents the computing resource needs of a DeterministicClient::WaitForNextActivation kRun cycle. **Tags:** atp.Status=draft

Table A.4: DeterministicClientResourceNeeds

Class	**Executable**
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure
Note	This meta-class represents an executable program. **Tags:** atp.Status=draft atp.recommendedPackage=Executables

▽

△

Class	**Executable**			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
buildType	BuildTypeEnum	0..1	attr	This attribute describes the buildType of a module and/or platform implementation.
minimumTimer Granularity	TimeValue	0..1	attr	This attribute describes the minimum timer resolution (TimeValue of one tick) that is required by the Executable. **Tags:** atp.Status=draft
rootSw Component Prototype	RootSwComponent Prototype	0..1	aggr	This represents the root SwCompositionPrototype of the Executable. This aggregation is required (in contrast to a direct reference of a SwComponentType) in order to support the definition of instanceRefs in Executable context. **Tags:** atp.Status=draft
version	StrongRevisionLabel String	0..1	attr	Version of the executable. **Tags:** atp.Status=draft

Table A.5: Executable

Class	**Machine**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::MachineManifest			
Note	Machine that represents an Adaptive Autosar Software Stack. **Tags:** atp.ManifestKind=MachineManifest atp.Status=draft atp.recommendedPackage=Machines			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
default Application Timeout	EnterExitTimeout	0..1	aggr	This aggration defines a default timeout in the context of a given Machine with respect to the launching and termination of applications. **Tags:** atp.Status=draft
environment Variable	TagWithOptionalValue	*	aggr	This aggregation represents the collection of environment variables that shall be added to the environment defined on the level of the enclosing Machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=environmentVariable atp.Status=draft
functionGroup	ModeDeclarationGroup Prototype	*	aggr	This aggregation represents the collection of function groups of the enclosing Machine. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel atp.Status=draft vh.latestBindingTime=preCompileTime
hwElement	HwElement	*	ref	This reference is used to describe the hardware resources of the machine. **Stereotypes:** atpUriDef **Tags:** atp.Status=draft

▽

△

Class	**Machine**			
machineDesign	MachineDesign	1	ref	Reference to the MachineDesign this Machine is implementing. **Tags:** atp.Status=draft
module Instantiation	AdaptiveModule Instantiation	*	aggr	Configuration of Adaptive Autosar module instances that are running on the machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=shortName atp.Status=draft
perState Timeout	PerStateTimeout	*	aggr	This aggregation represens the definition of per-state-timeouts in the context of the enclosing machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=perStateTimeout atp.Status=draft
processor	Processor	1..*	aggr	This represents the collection of processors owned by the enclosing machine. **Tags:** atp.Status=draft
secure Communication Deployment	SecureCommunication Deployment	*	aggr	Deployment of secure communication protocol configuration settings to crypto module entities. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel atp.Status=draft

Table A.6: Machine

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. **Tags:** atp.ManifestKind=ExecutionManifest,MachineManifest			
Base	*ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
–	–	–	–	–

Table A.7: ModeDeclaration

Primitive	**NormalizedInstruction**
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ProcessDesign
Note	This meta-class is used to describe runtime budget needs on the target system within Deterministic Client::WaitForNextActivation cycles. NormalizedInstructions does not reflect the actual number of code instructions, but allows the description of comparative resource needs. NormalizedInstructions is used for configuration of computing resources at integration time. NormalizedInstruction = runtime in sec * clock frequency in Hz **Tags:** atp.Status=draft xml.xsd.customType=NORMALIZED-INSTRUCTION xml.xsd.pattern=[1-9][0-9]* xml.xsd.type=string

Table A.8: NormalizedInstruction

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class provides information required to execute the referenced executable. **Tags:** atp.ManifestKind=ExecutionManifest atp.Status=draft atp.recommendedPackage=Processes			
Base	*ARElement*, *ARObject*, *AbstractExecutionContext*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadablePackageElement*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
design	ProcessDesign	0..1	ref	This reference represents the identification of the design-time representation for the Process that owns the reference. **Tags:** atp.Status=draft
deterministic Client	DeterministicClient	0..1	ref	This reference adds further execution characteristics for deterministic clients. **Tags:** atp.Status=draft
executable	Executable	0..1	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef **Tags:** atp.Status=draft
logTraceDefault LogLevel	LogTraceDefaultLog LevelEnum	0..1	attr	This attribute allows to set the initial log reporting level for a logTraceProcessId (ApplicationId).
logTraceFile Path	UriString	0..1	attr	This attribute defines the destination file to which the logging information is passed.
logTraceLog Mode	LogTraceLogMode Enum	0..1	attr	This attribute defines the destination of log messages provided by the process.
logTrace ProcessDesc	String	0..1	attr	This attribute can be used to describe the logTrace ProcessId that is used in the log and trace message in more detail.
logTrace ProcessId	String	0..1	attr	This attribute identifies the process in the log and trace message (ApplicationId).
preMapping	Boolean	0..1	attr	This attribute describes whether the executable is preloaded into the memory.
processState Machine	ModeDeclarationGroup Prototype	0..1	aggr	Set of Process States that are defined for the process. **Tags:** atp.Status=draft
stateDependent StartupConfig	StateDependentStartup Config	*	aggr	Applicable startup configurations. **Tags:** atp.Status=draft

Table A.9: Process

Class	**StartupConfig**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class represents a reusable startup configuration for processes.. **Tags:** atp.ManifestKind=ExecutionManifest atp.Status=draft			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
environment Variable	TagWithOptionalValue	*	aggr	This aggregation represents the collection of environment variables that shall be added to the respective Process's environment prior to launch. **Tags:** atp.Status=draft

▽

△

Class	StartupConfig			
scheduling Policy	SchedulingPolicyKind Enum	0..1	attr	This attribute represents the ability to define the scheduling policy for the initial thread of the application.
scheduling Priority	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 A.10: StartupConfig

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

Table A.11: StartupOption

Class	StateDependentStartupConfig			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class defines the startup configuration for the process depending on a collection of machine states. **Tags:** atp.ManifestKind=ExecutionManifest atp.Status=draft			
Base	*ARObject*			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
execution Dependency	ExecutionDependency	*	aggr	This attribute defines that all processes that are referenced via the ExecutionDependency shall be launched and shall reach a certain ProcessState before the referencing process is started. **Tags:** atp.Status=draft
functionGroup State	ModeDeclaration	*	iref	This represent the applicable functionGroupMode. **Tags:** atp.Status=draft
resourceGroup	ResourceGroup	1	ref	Reference to an applicable resource group. **Tags:** atp.Status=draft
startupConfig	StartupConfig	1	ref	Reference to a reusable startup configuration with startup parameters. **Tags:** atp.Status=draft

Table A.12: StateDependentStartupConfig

Class	**TagWithOptionalValue**			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::TagWithOptionalValue			
Note	A tagged value is a combination of a tag (key) and a value that gives supplementary information that is attached to a model element. Please note that keys without a value are allowed. **Tags:** atp.ManifestKind=ServiceInstanceManifest			
Base	*ARObject*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
key	String	1	attr	Defines a key.
value	String	0..1	attr	Defines the corresponding value.

Table A.13: TagWithOptionalValue

B Interfaces to other Functional Clusters (informative)

B.1 Overview

AUTOSAR decided not to standardize interfaces which are exclusively used between Functional Clusters (on platform-level only), to allow efficient implementations, which might depend e.g. on the used Operating System.

This chapter provides informative guidelines how the interaction between Functional Clusters looks like, by clustering the relevant requirements of this document to describe Inter-Functional Cluster (IFC) interfaces. In addition, the standardized public interfaces which are accessible by user space applications (see chapters 8 and 9) can also be used for interaction between Functional Clusters.

The goal is to provide a clear understanding of Functional Cluster boundaries and interaction, without specifying syntactical details. This ensures compatibility between documents specifying different Functional Clusters and supports parallel implementation of different Functional Clusters. Details of the interfaces are up to the platform provider. Additional interfaces, parameters and return values can be added.

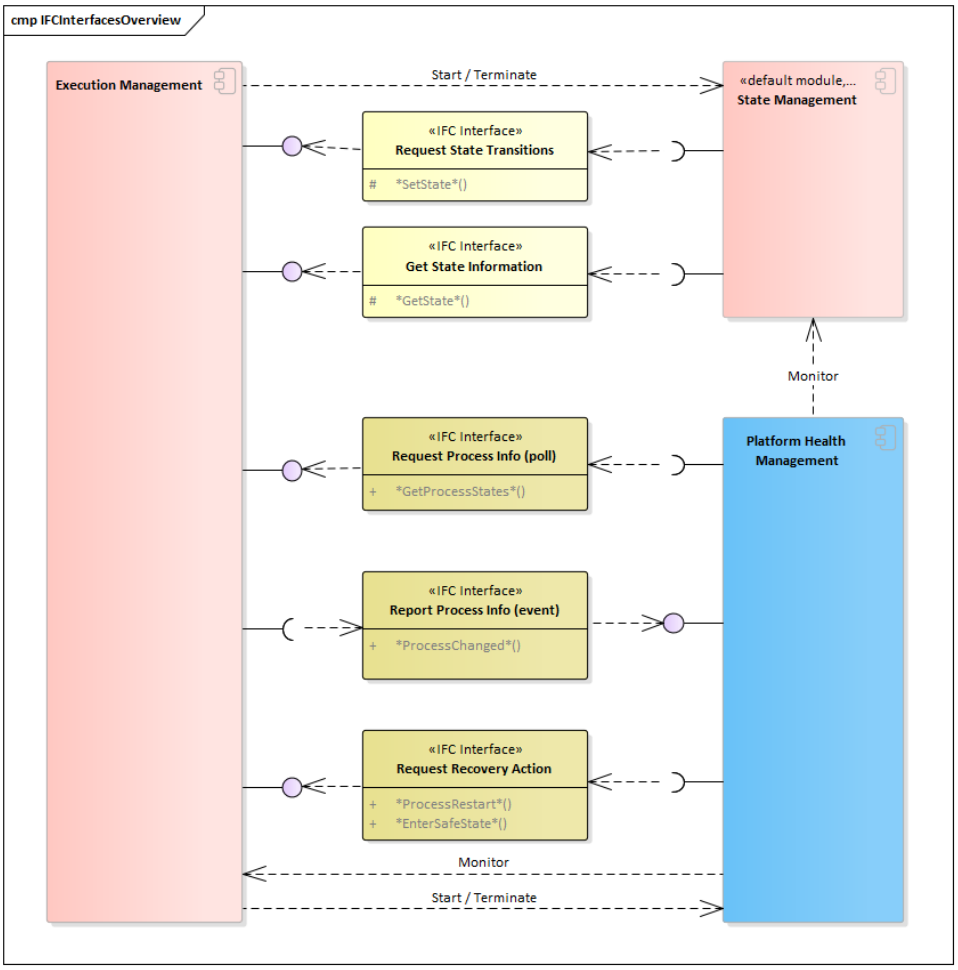

Figure B.1: Interfaces between Functional Clusters

B.2 Interface Tables

B.2.1 State Transition Request

	Name	Description	Requirements
Intended users	State Management		
Name proposal	*SetState*		
Functionality	Requests a change of Function Group States	The state change request shall lead to one or several state transitions and hereof state changes to the requested Function Group States	[SWS_EM_01026] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01068]
Parameters (in)	Function Group	Identifier of Function Group as defined in Machine Manifest (use identifier for “MachineState” to request a Machine State).	[SWS_EM_01107]

	State	Requested state of the Function Group. States are defined in the Machine Manifest. 1..* pairs of <Function Group><State> can be requested atomically.	[SWS_EM_01032] [SWS_EM_01107]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_02057]
	Execution Management is busy and cannot accept request	State change requests, that are received before all previously requested Function Group State transitions are completed	[SWS_EM_01034]
	State change request could not be finished in time	Timeout detected at state transition	[SWS_EM_02058]
	general error	[SWS_EM_02056]	

Table B.1: State Transition Request

B.2.2 Provide State Information

	Name	Description	Requirements
Intended users	State Management		
Name proposal	*GetState*		
Functionality	Get information about current state	The Execution Management provides an interface to retrieve the current Function Group State.	[SWS_EM_01028]
Parameters (in)	Function Group	Identifier of Function Group as defined in Machine Manifest (use identifier for “MachineState” to retrieve the Machine State).	[SWS_EM_01107]
Parameters (inout)	None		
Parameters (out)	State	Current Function Group State of the given Function Group. Empty if retrieval operation was not successful.	[SWS_EM_01032] [SWS_EM_01107]
Return value	Operation succeeded		[SWS_EM_02050]
	Execution Management is busy and cannot provide requested information	Execution Management performs a State transition of the requested Function Group	[SWS_EM_02044]
	general error	A state transition of the requested Function Group failed	[SWS_EM_02049]

Table B.2: Provide State Information

B.2.3 Get Process States Information

	Name	Description	Requirements
Intended users	Platform Health Management		
Name proposal	*GetProcessStates*		

Functionality	Get information about currently running processes	The Execution Management provides an interface to receive a list of the currently running processes. With this information the Platform Health Management can identify, based on Manifest information, if an how each process should be monitored. This polling API could be e.g. called once after startup and maintained by the information received via the reporting API specified in Platform Health Management.	[SWS_EM_02076]
Parameters (in)	None		
Parameters (inout)	None		
Parameters (out)	Processes	List of Currently running processes.	
Return value	Operation succeeded		
	Execution Management is busy and cannot provide requested information		
	general error		

Table B.3: Get Process States Information

B.2.4 Enter Safe State Request

	Name	Description	Requirements
Intended users	Platform Health Management	This is a "Recovery Action"	
Name proposal	*EnterSafeState*		
Functionality	Requests a change to a Safe State and stops all currently “ongoing” state transitions	The state change request shall immediately lead to a state transitions to the requested Safe State	[SWS_EM_01018] [SWS_EM_01061]
Parameters (in)	None		
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded	An Error is No Option as this is the final Safe State	

Table B.4: Enter Safe State Request

B.2.5 Process Restart Request

	Name	Description	Requirements
Intended users	Platform Health Management	This is a "Recovery Action"	
Name proposal	*ProcessRestart*		
Functionality	Request to restart a Process	Restart a specific Process on the request from the Platform Health Management.	[SWS_EM_01016] [SWS_EM_01062]

Parameters (in)	Process identifier	Unique named identifier of the Process to be restarted. Not the PID because this will change.	[SWS_EM_01016]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_01064]
	general error	Process could not be restarted	[SWS_EM_01063]

Table B.5: Process Restart Request

C History of Constraints and Specification Items

Please note that the lists in this chapter also include constraints and specification items that have been removed from the specification in a later version. These constraints and specification items do not appear as hyperlinks in the document.

C.1 Constraint and Specification Item History of this document according to AUTOSAR Release 17-10

C.1.1 Added Traceables in 17-10

Number	Heading
[SWS_EM_01001]	Execution Dependency error
[SWS_EM_01016]	RestartProcess API
[SWS_EM_01018]	`OverrideState` API
[SWS_EM_01032]	Machine States
[SWS_EM_01061]	OverrideState API interrupt
[SWS_EM_01062]	RestartProcess behaviour
[SWS_EM_01107]	Function Group name
[SWS_EM_01108]	Function Group State
[SWS_EM_01109]	State References
[SWS_EM_01110]	Off States
[SWS_EM_01111]	No reference to Off State
[SWS_EM_01112]	StartupConfig
[SWS_EM_01201]	Core Binding
[SWS_EM_02041]	ResetCause Enumeration
[SWS_EM_02042]	ApplicationClient::SetLastResetCause API
[SWS_EM_02043]	ApplicationClient::GetLastResetCause API

▽

△

Number	Heading
[SWS_EM_02044]	Machine State change in progress
[SWS_EM_02047]	StateClient::GetState API
[SWS_EM_02048]	Function Group State change in progress
[SWS_EM_02049]	State change failed
[SWS_EM_02050]	State change successful
[SWS_EM_02051]	Machine State change in progress
[SWS_EM_02054]	StateClient::SetState API
[SWS_EM_02055]	Function Group State change in progress
[SWS_EM_02056]	State change failed
[SWS_EM_02057]	State change successful
[SWS_EM_02070]	ApplicationReturnType Enumeration
[SWS_EM_02071]	
[SWS_EM_02072]	Retrieving Machine State
[SWS_EM_02073]	Retrieving Function Group State
[SWS_EM_02074]	Setting Machine State
[SWS_EM_02075]	Setting Function Group State
[SWS_EM_NA]	

Table C.1: Added Traceables in 17-10

C.1.2 Changed Traceables in 17-10

Number	Heading
[SWS_EM_01000]	Startup order
[SWS_EM_01002]	Idle Process State
[SWS_EM_01003]	Starting Process State
[SWS_EM_01004]	Running Process State
[SWS_EM_01005]	Terminating Process State
[SWS_EM_01006]	Terminated Process State
[SWS_EM_01012]	Application Argument Passing
[SWS_EM_01013]	Machine State and Function Group State
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01017]	Application Binary Name
[SWS_EM_01023]	Machine State Startup
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_01025]	Machine State Restart

▽

△

Number	Heading
[SWS_EM_01026]	State change
[SWS_EM_01028]	GetState API
[SWS_EM_01030]	Start of Application execution
[SWS_EM_01033]	Application start-up configuration
[SWS_EM_01034]	Deny State change request
[SWS_EM_01035]	Machine State Restart behavior
[SWS_EM_01036]	Machine State Shutdown behavior
[SWS_EM_01037]	Machine State Startup behavior
[SWS_EM_01039]	Scheduling priority range for SCHED_FIFO and SCHED_RR
[SWS_EM_01040]	Scheduling priority range for SCHED_OTHER
[SWS_EM_01041]	Scheduling FIFO
[SWS_EM_01042]	Scheduling Round-Robin
[SWS_EM_01043]	Scheduling Other
[SWS_EM_01050]	Start dependent Application Executables
[SWS_EM_01051]	Shutdown Application Executables
[SWS_EM_01053]	Application State Running
[SWS_EM_01055]	Application State Termination
[SWS_EM_01056]	State Manager
[SWS_EM_01058]	Shutdown of the Operating System
[SWS_EM_01059]	Restart of the Operating System
[SWS_EM_01060]	State change behavior
[SWS_EM_02000]	ApplicationState Enumeration
[SWS_EM_02001]	
[SWS_EM_02002]	ApplicationClient::~ApplicationClient API
[SWS_EM_02003]	ApplicationClient::ReportApplicationState API
[SWS_EM_02005]	StateReturnType Enumeration
[SWS_EM_02006]	
[SWS_EM_02007]	StateClient::StateClient API
[SWS_EM_02008]	StateClient::~StateClient API
[SWS_EM_02030]	ApplicationClient::ApplicationClient API
[SWS_EM_02031]	Application State Reporting

Table C.2: Changed Traceables in 17-10

C.1.3 Deleted Traceables in 17-10

Number	Heading
[SWS_EM_00017]	Application Processes
[SWS_EM_01027]	Rejection of Client Requests
[SWS_EM_01029]	`SetMachineState` API
[SWS_EM_01052]	Application State `Initializing`
[SWS_EM_01057]	Machine State Change arbitration
[SWS_EM_02009]	
[SWS_EM_02014]	
[SWS_EM_02019]	
[SWS_EM_99999]	

Table C.3: Deleted Traceables in 17-10

C.1.4 Added Constraints in 17-10

none

C.1.5 Changed Constraints in 17-10

none

C.1.6 Deleted Constraints in 17-10

none

C.2 Constraint and Specification Item History of this document according to AUTOSAR Release 18-03

C.2.1 Added Traceables in 18-03

Number	Heading
[SWS_EM_01044]	Machine States Identification
[SWS_EM_01063]	Process Restart Failed
[SWS_EM_01064]	Process Restart Successful
[SWS_EM_01065]	Shutdown state timeout monitoring behavior

▽

△

Number	Heading
[SWS_EM_01066]	Start state change behavior
[SWS_EM_01067]	Confirm State Changes
[SWS_EM_01068]	Report start-up timeout
[SWS_EM_01069]	Self-terminating Process State
[SWS_EM_01070]	Acknowledgement of termination request
[SWS_EM_01071]	Initiation of Process self-termination
[SWS_EM_01072]	Application Argument Zero
[SWS_EM_01073]	Simple Arguments
[SWS_EM_01074]	Short form arguments with option value
[SWS_EM_01075]	Short form Arguments without option value
[SWS_EM_01076]	Long form Arguments with option value
[SWS_EM_01077]	Long form Arguments without option value
[SWS_EM_01301]	Cyclic Execution
[SWS_EM_01302]	Cyclic Execution Control
[SWS_EM_01305]	Worker Pool
[SWS_EM_01308]	Random Numbers
[SWS_EM_01310]	Get Activation Time
[SWS_EM_01311]	Activation Time Unknown
[SWS_EM_01312]	Get Next Activation Time
[SWS_EM_01313]	Next Activation Time Unknown
[SWS_EM_02058]	State Transition Timeout
[SWS_EM_02102]	Memory control
[SWS_EM_02103]	CPU usage control
[SWS_EM_02104]	Core affinity
[SWS_EM_02106]	ResourceGroup assignment
[SWS_EM_02107]	Maximum heap
[SWS_EM_02108]	Maximum system memory usage
[SWS_EM_02109]	Process pre-mapping
[SWS_EM_02201]	ActivationReturnType Enumeration
[SWS_EM_02202]	ActivationTimeStampReturnType Enumeration
[SWS_EM_02210]	
[SWS_EM_02211]	DeterministicClient::DeterministicClient API
[SWS_EM_02215]	DeterministicClient::~DeterministicClient API
[SWS_EM_02216]	DeterministicClient::WaitForNextActivation API
[SWS_EM_02220]	DeterministicClient::RunWorkerPool API
[SWS_EM_02225]	DeterministicClient::GetRandom API
[SWS_EM_02230]	DeterministicClient::GetActivationTime API

▽

△

Number	Heading
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime API

Table C.4: Added Traceables in 18-03

C.2.2 Changed Traceables in 18-03

Number	Heading
[SWS_EM_01000]	Startup order
[SWS_EM_01001]	Execution Dependency error
[SWS_EM_01002]	Idle Process State
[SWS_EM_01003]	Starting Process State
[SWS_EM_01004]	Running Process State
[SWS_EM_01005]	Terminating Process State
[SWS_EM_01006]	Terminated Process State
[SWS_EM_01012]	Application Argument Passing
[SWS_EM_01013]	Machine State and Function Group State
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01016]	Restart Process
[SWS_EM_01018]	Override State
[SWS_EM_01023]	Machine State Startup
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_01025]	Machine State Restart
[SWS_EM_01026]	State Change
[SWS_EM_01028]	Get State Information
[SWS_EM_01030]	Start of Process execution
[SWS_EM_01032]	Machine States Obtainment
[SWS_EM_01033]	Application start-up configuration
[SWS_EM_01034]	Deny State Change Request
[SWS_EM_01035]	Machine State Restart behavior
[SWS_EM_01036]	Machine State Shutdown behavior
[SWS_EM_01037]	Machine State Startup behavior
[SWS_EM_01041]	Scheduling FIFO
[SWS_EM_01042]	Scheduling Round-Robin
[SWS_EM_01043]	Scheduling Other
[SWS_EM_01050]	Start Dependent Processes
[SWS_EM_01051]	Shutdown Processes

▽

△

Number	Heading
[SWS_EM_01053]	Application State Running
[SWS_EM_01055]	Initiation of Process termination
[SWS_EM_01058]	Shutdown of the Operating System
[SWS_EM_01059]	Restart of the Operating System
[SWS_EM_01060]	Shutdown state change behavior
[SWS_EM_01061]	Override State Interrupt
[SWS_EM_01062]	Restart Process Behavior
[SWS_EM_01107]	Function Group name
[SWS_EM_01108]	Function Group State
[SWS_EM_01109]	State References
[SWS_EM_01110]	Off States
[SWS_EM_02001]	
[SWS_EM_02044]	State Change in Progress
[SWS_EM_02049]	State Change Failed
[SWS_EM_02050]	State Information Success
[SWS_EM_02056]	State Change Failed
[SWS_EM_02057]	State Change Successful
[SWS_EM_NA]	

Table C.5: Changed Traceables in 18-03

C.2.3 Deleted Traceables in 18-03

Number	Heading
[SWS_EM_01017]	Application Binary Name
[SWS_EM_01056]	State Manager
[SWS_EM_01112]	StartupConfig
[SWS_EM_01201]	Core Binding
[SWS_EM_02005]	StateReturnType Enumeration
[SWS_EM_02006]	
[SWS_EM_02007]	StateClient::StateClient API
[SWS_EM_02008]	StateClient::~StateClient API
[SWS_EM_02031]	Application State Reporting
[SWS_EM_02041]	ResetCause Enumeration
[SWS_EM_02042]	ApplicationClient::SetLastResetCause API
[SWS_EM_02043]	ApplicationClient::GetLastResetCause API
[SWS_EM_02047]	StateClient::GetState API

▽

△

Number	Heading
[SWS_EM_02048]	Function Group State change in progress
[SWS_EM_02051]	Machine State change in progress
[SWS_EM_02054]	StateClient::SetState API
[SWS_EM_02055]	Function Group State change in progress
[SWS_EM_02071]	
[SWS_EM_02072]	Retrieving Machine State
[SWS_EM_02073]	Retrieving Function Group State
[SWS_EM_02074]	Setting Machine State
[SWS_EM_02075]	Setting Function Group State

Table C.6: Deleted Traceables in 18-03

C.2.4 Added Constraints in 18-03

none

C.2.5 Changed Constraints in 18-03

none

C.2.6 Deleted Constraints in 18-03

none

C.3 Constraint and Specification Item History of this document according to AUTOSAR Release 18-10

C.3.1 Added Traceables in 18-10

none

C.3.2 Changed Traceables in 18-10

Number	Heading
[SWS_EM_01000]	Startup order
[SWS_EM_01001]	Execution Dependency error
[SWS_EM_01004]	Running Process State
[SWS_EM_01005]	Terminating Process State
[SWS_EM_01012]	Process Argument Passing
[SWS_EM_01013]	Machine State and Function Group State
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01018]	Override State
[SWS_EM_01023]	Machine State Startup
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_01025]	Machine State Restart
[SWS_EM_01026]	State Change
[SWS_EM_01028]	Get State Information
[SWS_EM_01033]	Process start-up configuration
[SWS_EM_01034]	Deny State Change Request
[SWS_EM_01035]	Machine State Restart behavior
[SWS_EM_01036]	Machine State Shutdown behavior
[SWS_EM_01037]	Machine State Startup behavior
[SWS_EM_01039]	Scheduling priority range for SCHED_FIFO and SCHED_RR
[SWS_EM_01040]	Scheduling priority range for SCHED_OTHER
[SWS_EM_01041]	Scheduling FIFO
[SWS_EM_01042]	Scheduling Round-Robin
[SWS_EM_01043]	Scheduling Other
[SWS_EM_01053]	Execution State Running
[SWS_EM_01060]	Shutdown state change behavior
[SWS_EM_01065]	Shutdown state timeout monitoring behavior
[SWS_EM_01066]	Start state change behavior
[SWS_EM_01067]	Confirm State Changes
[SWS_EM_01069]	Self-terminating Process State
[SWS_EM_01070]	Acknowledgement of termination request
[SWS_EM_01071]	Initiation of Process self-termination
[SWS_EM_01072]	Process Argument Zero
[SWS_EM_01074]	Short form arguments with option value
[SWS_EM_01075]	Short form Arguments without option value
[SWS_EM_01076]	Long form Arguments with option value

▽

△

Number	Heading
[SWS_EM_01077]	Long form Arguments without option value
[SWS_EM_01107]	Function Group configuration
[SWS_EM_01109]	Misconfigured Process instances
[SWS_EM_01110]	Off States
[SWS_EM_02000]	ExecutionState Enumeration
[SWS_EM_02001]	
[SWS_EM_02002]	ExecutionClient::~ExecutionClient API
[SWS_EM_02003]	ExecutionClient::ReportExecutionState API
[SWS_EM_02030]	ExecutionClient::ExecutionClient API
[SWS_EM_02044]	State Change in Progress
[SWS_EM_02049]	State Change Failed
[SWS_EM_02070]	ExecutionReturnType Enumeration
[SWS_EM_02109]	Process pre-mapping
[SWS_EM_02210]	
[SWS_EM_NA]	

Table C.7: Changed Traceables in 18-10

C.3.3 Deleted Traceables in 18-10

Number	Heading
[SWS_EM_01044]	Machine States Identification
[SWS_EM_01108]	Function Group State
[SWS_EM_01111]	No reference to Off State

Table C.8: Deleted Traceables in 18-10

C.3.4 Added Constraints in 18-10

none

C.3.5 Changed Constraints in 18-10

none

C.3.6 Deleted Constraints in 18-10

none