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

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R24-11

Document Change History

Date	Release	Changed by	Description
2024-11-27	R24-11	AUTOSAR Release Management	• Requirements for deterministic execution are removed • Added support for Function Group access control • API refinement (define lifetime of arguments, error codes, removed thread-safety information, return values) • Added clarification on Execution Dependencies
2023-11-23	R23-11	AUTOSAR Release Management	• Requirements for deterministic execution are set to obsolete • The right to create child processes can be configured by integrator • Added support for standardized trace points • API Refinement (ExecutionClient termination handler, remove FunctionGroup, C++ Core Guidelines compliance) • Clarification of Unrecoverable State

△

2022-11-24	R22-11	AUTOSAR Release Management	• Clarification on error handling during Function Group State transition • Changes to `ara::exec::ExecErrc` • Clarification on interaction between Platform Health Management and Execution Management
2021-11-25	R21-11	AUTOSAR Release Management	• Clarified handling of unexpected Process termination • `ara::exec::StateClient` API updated (constructor token removed) • Invalid state transitions identified and handling defined • `ara::exec::DeterministicClient` API and behaviour clarified
2020-11-30	R20-11	AUTOSAR Release Management	• Further refinement of State Management API and semantics • Update process lifecycle (terminating report optional) • Added Deterministic Synchronization support • EM-PHM interaction
2019-11-28	R19-11	AUTOSAR Release	• Further refinement of State Management API and semantics • Introduced support for trusted platform • Added support for non-reporting Processes • Execution Management API uses Core types • Changed Document Status from Final to published
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

Contents

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.

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

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 Modelled Processes. Modelled Processes are self-contained, e.g. have internal control of thread creation. Execution Management performs these tasks using information contained within one or more Manifest content such as when and how Executables should be started. Execution Management also provides support for State Management (see Section 7.5) and Security (Section 7.8).

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 Section 8.

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. Therefore, Execution Management has no standardized dependencies. The launching order of applications 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 [1] 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

The glossary below includes acronyms and abbreviations relevant to this document.

Term	Description
Reporting Process	A type of Modelled Process with an associated Executable where reportingBehavior is omitted ([TPS_MANI_01279] [2]) or set to reportsExecutionState. A Reporting Process is expected to report its Execution State to Execution Management.
Non-reporting Process	A type of Modelled Process with an associated Executable where reportingBehavior is set to doesNotReportExecutionState ([TPS_MANI_01279] [2]). Please note that the *Non-reporting Process* was **not** developed for the AUTOSAR Adaptive Platform and for this reason it is not using Adaptive Platform Services. For example, it is not using ara::exec::ExecutionClient API and therefore cannot report *Execution State* to Execution Management. In general, this type of a process is intended to allow integration of a software that cannot be modified for use with the AUTOSAR Adaptive Platform.
Companion Process	A type of Reporting Process that is associated with Non-reporting Process and used to determine when functionality expected from Non-reporting Process is available. Whenever functional dependencies on Non-reporting Processes exist, the integrator can configure proxy Execution Dependencies on the Companion Process and make the Companion Process kRunning reporting conditional on monitored Non-reporting Process.
Self-terminating Process	A type of Modelled Process that has terminationBehavior configured to processIsSelfTerminating. This type of Modelled Process is allowed to self initiate termination procedure (i.e. just terminate with exit status EXIT_SUCCESS), or wait for Execution Management to initiate termination procedure via SIGTERM.
Unexpected Self-termination	The event consumed by Execution Management when a Modelled Process terminates without justified reason, for example: • termination without prior request where terminationBehavior is configured to processIsNotSelfTerminating. • termination before reporting kRunning. Please note that every Unexpected Self-termination is also an Unexpected Termination, so requirements for the later apply here as well.
Unexpected Termination	The event consumed by Execution Management when a Modelled Process terminates with exit status other than 0 (EXIT_SUCCESS). Any kind of unhandled signal will result in an Unexpected Termination and thus a non 0 exit status.

Undefined Function Group State	Any state of a `Function Group`, which is not modelled. A `Function Group` is in an `Undefined Function Group State` during state transition, if a state transition failed or if an `Unexpected Termination` or `Unexpected Self-termination` happened.
StateClient	`State Management` interface to `Execution Management` to support `Function Group State` and `Machine State` management.
Unrecoverable State	A state entered by `Execution Management` in response to a situation that it cannot resolve. In the state, `Execution Management` stops taking any further actions, terminates all processes managed by `Execution Management` and provides a facility for further project-specific handling.

Table 2.1: Technical Terms

The following technical terms used in this document are defined in the corresponding document mentioned in the table below.

Term	Description
Communication Management	see [1] Specification of Communication Management
PLATFORM_CORE	see [2] TPS Manifest Specification
Modelled Process	see [3] Requirements on Execution Management
Execution Dependency	see [3] Requirements on Execution Management
Execution Management	see [3] Requirements on Execution Management
Function Group	see [3] Requirements on Execution Management
Function Group State	see [3] Requirements on Execution Management
Machine State	see [3] Requirements on Execution Management
Process State	see [3] Requirements on Execution Management
Operating System	see [4] Requirements on Operating System Interface
State Management	see [5] Requirements of State Management
Execution Manifest	see [6] Methodology for Adaptive Platform
Machine Manifest	see [6] Methodology for Adaptive Platform
Service Instance Manifest	see [6] Methodology for Adaptive Platform
Platform Health Management	see [7] Specification of Platform Health Management
Adaptive Application	see [8] AUTOSAR Glossary
AUTOSAR Adaptive Platform	see [8] AUTOSAR Glossary
Adaptive Platform Foundation	see [8] AUTOSAR Glossary
Adaptive Platform Services	see [8] AUTOSAR Glossary
Manifest	see [8] AUTOSAR Glossary
Executable	see [8] AUTOSAR Glossary
Functional Cluster	see [8] AUTOSAR Glossary
Machine	see [8] AUTOSAR Glossary
Processed Manifest	see [8] AUTOSAR Glossary
Process	see [8] AUTOSAR Glossary
Service	see [8] AUTOSAR Glossary
Service Interface	see [8] AUTOSAR Glossary
Service Discovery	see [8] AUTOSAR Glossary
Trusted Platform	see [8] AUTOSAR Glossary

Table 2.2: Reference to Technical Terms

3 Related documentation

3.1 Input documents & related standards and norms

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

[1] Specification of Communication Management
AUTOSAR_AP_SWS_CommunicationManagement

[2] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[3] Requirements on Execution Management
AUTOSAR_AP_RS_ExecutionManagement

[4] Requirements on Operating System Interface
AUTOSAR_AP_RS_OperatingSystemInterface

[5] Requirements of State Management
AUTOSAR_AP_RS_StateManagement

[6] Methodology for Adaptive Platform
AUTOSAR_AP_TR_Methodology

[7] Specification of Platform Health Management
AUTOSAR_AP_SWS_PlatformHealthManagement

[8] Glossary
AUTOSAR_FO_TR_Glossary

[9] Specification of Adaptive Platform Core
AUTOSAR_AP_SWS_Core

[10] Explanation of Adaptive Platform Software Architecture
AUTOSAR_AP_EXP_SWArchitecture

[11] Specification of State Management
AUTOSAR_AP_SWS_StateManagement

[12] Safety Requirements for AUTOSAR Adaptive Platform and AUTOSAR Classic Platform
AUTOSAR_FO_RS_Safety

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

[14] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, 'Basic Concepts and Taxonomy of Dependable and Secure Computing', IEEE Transac-

tions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004

[15] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[16] Explanation of Identity and Access Management
AUTOSAR_AP_EXP_IdentityAndAccessManagement

[17] Specification of Execution Management
AUTOSAR_AP_SWS_ExecutionManagement

3.2 Further applicable specification

AUTOSAR provides a core specification [9] which is also applicable for Execution Management. The chapter “General requirements for all FunctionalClusters” of this specification shall be considered as an additional and required specification for implementation of Execution Management.

4 Constraints and assumptions

4.1 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 7.6 Resource Limitation and Section 7.7 Fault Tolerance 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.
- Thread safety and reentrancy of ara::exec API was not considered in this release.

Section 6.1 details requirements from Execution Management Requirement Specification [3] 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.

5 Dependencies to other Functional Clusters

This chapter provides an overview of the dependencies to other Functional Clusters in the AUTOSAR Adaptive Platform. Section 5.1 “Provided Interfaces” lists the interfaces provided by `Execution Management` to other Functional Clusters. Section 5.2 “Required Interfaces” lists the interfaces required by `Execution Management`.

A detailed technical architecture documentation of the AUTOSAR Adaptive Platform is provided in [10].

5.1 Provided Interfaces

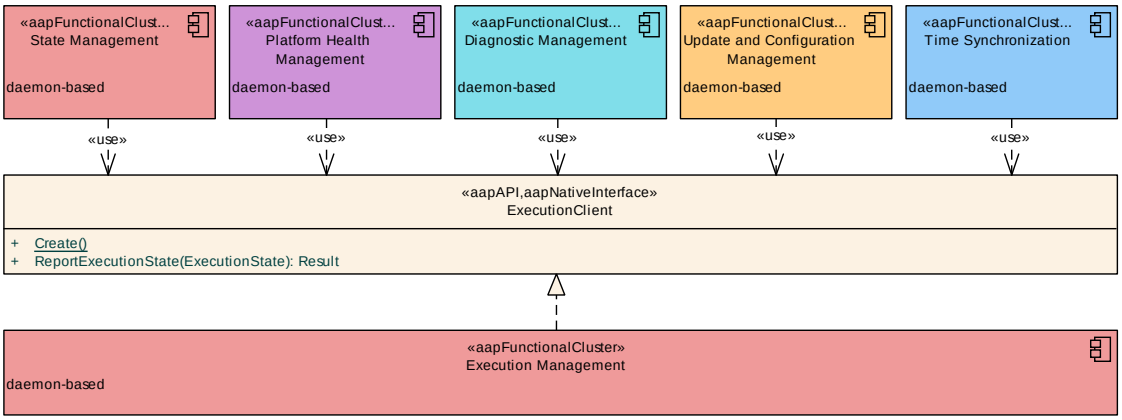

Figure 5.1: Interfaces provided by Execution Management to other Functional Clusters

Figure 5.1 shows the interfaces provided by `Execution Management` to other Functional Clusters within the AUTOSAR Adaptive Platform.

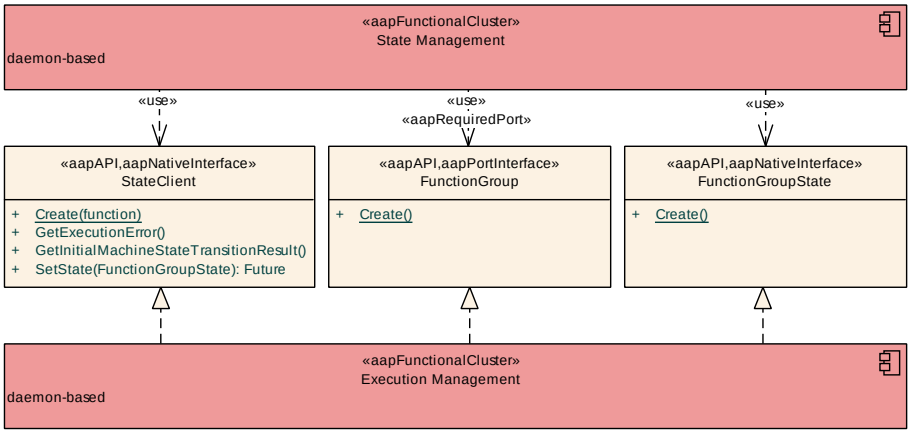

Figure 5.2: Interfaces provided by Execution Management to State Management

Figure 5.2 shows the interfaces provided by `Execution Management` to `State Management`. Table 5.1 provides a complete list of interfaces provided to other Functional Clusters within the AUTOSAR Adaptive Platform.

Interface	*Functional Cluster*	*Purpose*
ExecutionClient	Diagnostic Management	This interface is used to report the status of the Diagnostic Management daemon process(es).
	Platform Health Management	Platform Health Management uses this interface to report the state of its daemon process to Execution Management.
	State Management	This interface shall be used to report the state of the State Management process(es).
	Time Synchronization	Time Synchronization shall use this interface to report the state of its daemon process.
	Update and Configuration Management	This interface shall be used by the daemon process(es) inside Update and Configuration Management to report their execution state to Execution Management.
	Vehicle Update and Configuration Management	This interface shall be used by the daemon process(es) inside Vehicle Update and Configuration Management to report their execution state to Execution Management.
FunctionGroup	State Management	This interface shall be used to construct FunctionGroupStates.
FunctionGroup State	State Management	This interface shall be used to request FunctionGroupState transitions.
StateClient	State Management	This interface shall be used to request FunctionGroupState transitions.

Table 5.1: Interfaces provided to other Functional Clusters

5.2 Required Interfaces

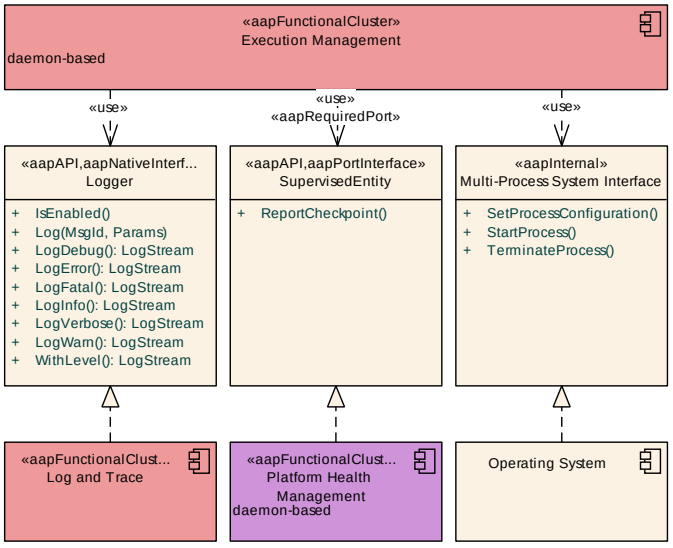

Figure 5.3: Interfaces required by Execution Management from other Functional Clusters

Figure 5.3 shows the interfaces required by Execution Management from other Functional Clusters within the AUTOSAR Adaptive Platform. Table 5.2 provides a complete list of required interfaces from other Functional Clusters within the AUTOSAR Adaptive Platform.

Functional Cluster	*Interface*	*Purpose*
Log and Trace	Logger	`Execution Management` shall use this interface to log standardized messages.
Platform Health Management	SupervisedEntity	`Execution Management` shall use this interface to enable supervision of its process(es) by `Platform Health Management`.

Table 5.2: Interfaces required from other Functional Clusters

6 Requirements Tracing

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

Requirement	Description	Satisfied by
[RS_AP_00114]	C++ interface shall be compatible with C++14	[SWS_EM_02560] [SWS_EM_02561]
[RS_AP_00116]	Header file name	[SWS_EM_02544]
[RS_AP_00119]	Return values / application errors	[SWS_EM_02003] [SWS_EM_02276] [SWS_EM_02278] [SWS_EM_02279] [SWS_EM_02560] [SWS_EM_02561] [SWS_EM_02562]
[RS_AP_00120]	Method and Function names	[SWS_EM_02003] [SWS_EM_02276] [SWS_EM_02278] [SWS_EM_02279] [SWS_EM_02283] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02290] [SWS_EM_02291] [SWS_EM_02542] [SWS_EM_02560] [SWS_EM_02561] [SWS_EM_02562]
[RS_AP_00121]	Parameter names	[SWS_EM_02003] [SWS_EM_02276] [SWS_EM_02278] [SWS_EM_02283] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02291] [SWS_EM_02542] [SWS_EM_02560] [SWS_EM_02561]
[RS_AP_00122]	Type names	[SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284] [SWS_EM_02541] [SWS_EM_02544]
[RS_AP_00124]	Variable names	[SWS_EM_02544] [SWS_EM_02545] [SWS_EM_02546]
[RS_AP_00125]	Enumerator and constant names	[SWS_EM_02000] [SWS_EM_02281]
[RS_AP_00127]	Usage of ara::core types	[SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284]
[RS_AP_00128]	Error reporting	[SWS_EM_02003] [SWS_EM_02278] [SWS_EM_02279] [SWS_EM_02542] [SWS_EM_02562]
[RS_AP_00129]	Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation	[SWS_EM_02000] [SWS_EM_02269] [SWS_EM_02281]
[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] [SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02283] [SWS_EM_02284] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02290] [SWS_EM_02291]
[RS_AP_00134]	noexcept behavior of class destructors	[SWS_EM_02000] [SWS_EM_02002] [SWS_EM_02272] [SWS_EM_02277]
[RS_AP_00137]	Connecting run-time interface with model	[SWS_EM_02586]
[RS_AP_00138]	Return type of asynchronous function calls	[SWS_EM_02278] [SWS_EM_02279]
[RS_AP_00139]	Return type of synchronous function calls	[SWS_EM_02003] [SWS_EM_02276] [SWS_EM_02542] [SWS_EM_02562]

△

Requirement	Description	Satisfied by
[RS_AP_00140]	Usage of "final specifier"	[SWS_EM_02282] [SWS_EM_02544]
[RS_AP_00142]	Handling of unsuccessful operations	[SWS_EM_02281]
[RS_AP_00143]	Use 32-bit integral types by default	[SWS_EM_02000]
[RS_AP_00144]	Availability of a named constructor	[SWS_EM_02276] [SWS_EM_02562]
[RS_AP_00145]	Availability of special member functions	[SWS_EM_02002] [SWS_EM_02272] [SWS_EM_02277] [SWS_EM_02325] [SWS_EM_02330] [SWS_EM_02331] [SWS_EM_02332] [SWS_EM_02563] [SWS_EM_02564] [SWS_EM_02565] [SWS_EM_02566] [SWS_EM_02567] [SWS_EM_02568] [SWS_EM_02580] [SWS_EM_02581]
[RS_AP_00147]	Classes that are created with an InstanceSpecifier as an argument are not copyable, but at most movable.	[SWS_EM_02322] [SWS_EM_02327]
[RS_AP_00149]	Error handling for non-initialized Functional Cluster	[SWS_EM_02281] [SWS_EM_02557]
[RS_AP_00150]	Provide only interfaces that are intended to be used by AUTOSAR Applications and Functional Clusters	[SWS_EM_02001] [SWS_EM_02263] [SWS_EM_02269] [SWS_EM_02275] [SWS_EM_02282] [SWS_EM_02284]
[RS_AP_00151]	C++ Core Guidelines	[SWS_EM_02331] [SWS_EM_02332] [SWS_EM_02560] [SWS_EM_02561] [SWS_EM_02566] [SWS_EM_02567] [SWS_EM_02580] [SWS_EM_02581]
[RS_AP_00153]	Assignment operators should restrict "this" to lvalues	[SWS_EM_02330] [SWS_EM_02332]
[RS_AP_00154]	Internal namespaces	[SWS_EM_02001] [SWS_EM_02263] [SWS_EM_02269] [SWS_EM_02275] [SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284] [SWS_EM_02290] [SWS_EM_02291] [SWS_EM_02541] [SWS_EM_02544]
[RS_AP_00155]	Avoidance of cluster-specific initialization functions	[SWS_EM_02557]
[RS_AP_00156]	Naming conventions for L&T Context ID	[SWS_EM_02569] [SWS_EM_02570] [SWS_EM_02571] [SWS_EM_02572]
[RS_EM_00002]	Execution Management shall set-up one process for the execution of each Modelled 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_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]	Execution Management shall control the right to create child process for each process it starts.	[SWS_EM_01033] [SWS_EM_02559]
[RS_EM_00010]	Execution Management shall support multiple instances of Executables.	[SWS_EM_01012] [SWS_EM_01072] [SWS_EM_01078] [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249]

▽

△

Requirement	Description	Satisfied by
[RS_EM_00011]	Execution Management shall support self-initiated graceful shutdown of processes.	[SWS_EM_01006] [SWS_EM_01404]
[RS_EM_00014]	Execution Management shall support a Trusted Platform.	[SWS_EM_02299] [SWS_EM_02300] [SWS_EM_02301] [SWS_EM_02302] [SWS_EM_02303] [SWS_EM_02305] [SWS_EM_02306] [SWS_EM_02307] [SWS_EM_02308] [SWS_EM_02309] [SWS_EM_02556]
[RS_EM_00015]	Execution Management shall support integrity and authenticity monitoring.	[SWS_EM_02300] [SWS_EM_02301] [SWS_EM_02302] [SWS_EM_02303] [SWS_EM_02305] [SWS_EM_02306] [SWS_EM_02400] [SWS_EM_02556]
[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_CONSTR_00001] [SWS_EM_CONSTR_01744]
[RS_EM_00101]	Execution Management shall support State Management functionality.	[SWS_EM_01023] [SWS_EM_01032] [SWS_EM_01033] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01110] [SWS_EM_02241] [SWS_EM_02245] [SWS_EM_02250] [SWS_EM_02251] [SWS_EM_02253] [SWS_EM_02255] [SWS_EM_02258] [SWS_EM_02259] [SWS_EM_02260] [SWS_EM_02263] [SWS_EM_02266] [SWS_EM_02267] [SWS_EM_02268] [SWS_EM_02269] [SWS_EM_02272] [SWS_EM_02273] [SWS_EM_02274] [SWS_EM_02275] [SWS_EM_02276] [SWS_EM_02277] [SWS_EM_02278] [SWS_EM_02279] [SWS_EM_02280] [SWS_EM_02295] [SWS_EM_02296] [SWS_EM_02297] [SWS_EM_02298] [SWS_EM_02310] [SWS_EM_02312] [SWS_EM_02315] [SWS_EM_02316] [SWS_EM_02321] [SWS_EM_02322] [SWS_EM_02324] [SWS_EM_02325] [SWS_EM_02327] [SWS_EM_02328] [SWS_EM_02329] [SWS_EM_02330] [SWS_EM_02331] [SWS_EM_02332] [SWS_EM_02541] [SWS_EM_02542] [SWS_EM_02543] [SWS_EM_02544] [SWS_EM_02545] [SWS_EM_02546] [SWS_EM_02549] [SWS_EM_02552] [SWS_EM_02555] [SWS_EM_02561] [SWS_EM_02583] [SWS_EM_02584] [SWS_EM_02585] [SWS_EM_02586] [SWS_EM_CONSTR_02556] [SWS_EM_CONSTR_02557] [SWS_EM_CONSTR_02558] [SWS_EM_CONSTR_02559] [SWS_EM_CONSTR_02560]

▽

△

Requirement	Description	Satisfied by
[RS_EM_00103]	Execution Management shall support process lifecycle management.	[SWS_EM_01002] [SWS_EM_01003] [SWS_EM_01004] [SWS_EM_01006] [SWS_EM_01055] [SWS_EM_01210] [SWS_EM_01211] [SWS_EM_01212] [SWS_EM_01309] [SWS_EM_01314] [SWS_EM_01401] [SWS_EM_01402] [SWS_EM_01403] [SWS_EM_01404] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02243] [SWS_EM_02558] [SWS_EM_02560] [SWS_EM_02562] [SWS_EM_02563] [SWS_EM_02564] [SWS_EM_02565] [SWS_EM_02566] [SWS_EM_02567] [SWS_EM_02568] [SWS_EM_02577] [SWS_EM_02578] [SWS_EM_02579] [SWS_EM_02580] [SWS_EM_02581] [SWS_EM_02582]
[RS_EM_00111]	Execution Management shall assist identification of processes during Machine runtime.	[SWS_EM_02400]
[RS_EM_00150]	Error Handling.	[SWS_EM_02032] [SWS_EM_02033] [SWS_EM_02034] [SWS_EM_02547] [SWS_EM_02548]
[RS_EM_00152]	Execution Management shall support standardized trace points throughout the state transitions.	[SWS_EM_02569] [SWS_EM_02570] [SWS_EM_02571] [SWS_EM_02572] [SWS_EM_02573] [SWS_EM_02574] [SWS_EM_02575] [SWS_EM_02576]

Table 6.1: Requirements Tracing

6.1 Not applicable requirements

[SWS_EM_NA]

Upstream requirements: RS_AP_00111, RS_AP_00115, RS_AP_00135, RS_AP_00148, RS_AP_00141, RS_AP_00146, RS_EM_00151, RS_EM_NA

⌈These requirements are not applicable as they are not within the scope of this release.⌋

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 7.2 presents an introduction to key terms within Execution Management focusing on the relationship between application, Executable, and Modelled Process. With the latter, we refer to an instance of the meta-model describing a process, it will eventually be realized by an operating system Process.
- Section 7.3 covers the core Execution Management run-time responsibilities including the start of applications.
- Section 7.4 describes the lifecycle of applications including Modelled Process state transitions and startup / shutdown sequences.
- Section 7.5 covers several topics related to State Management within Execution Management including Function Group state management and state transition behavior.
- Section 7.6 describes how Execution Management supports resource management including the limitation of usage of CPU and memory by an application.
- Section 7.7 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.
- Section 7.8 covers the topic of Trusted Platform, i.e. ensuring the integrity and authenticity of applications.

7.1 Functional Cluster Lifecyle

7.1.1 Startup

See Section 7.5.2.1.

7.1.2 Shutdown

See Section 7.5.2.2.

7.1.3 Restart

See Section 7.5.2.2.

7.2 Technical Overview

This chapter presents a short summary of the relationship between application, `Executable`, and `Modelled Process`.

7.2.1 Executable

An `Executable` is a software unit which is part of an application. It has exactly one entry point (main function) [SWS_OSI_01001]. An application can be implemented in one or more `Executables` [TPS_MANI_01010] [2].

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 Section 7.2.3 and [2], 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 7.1 shows the lifecycle of an Executable from deployment to execution.

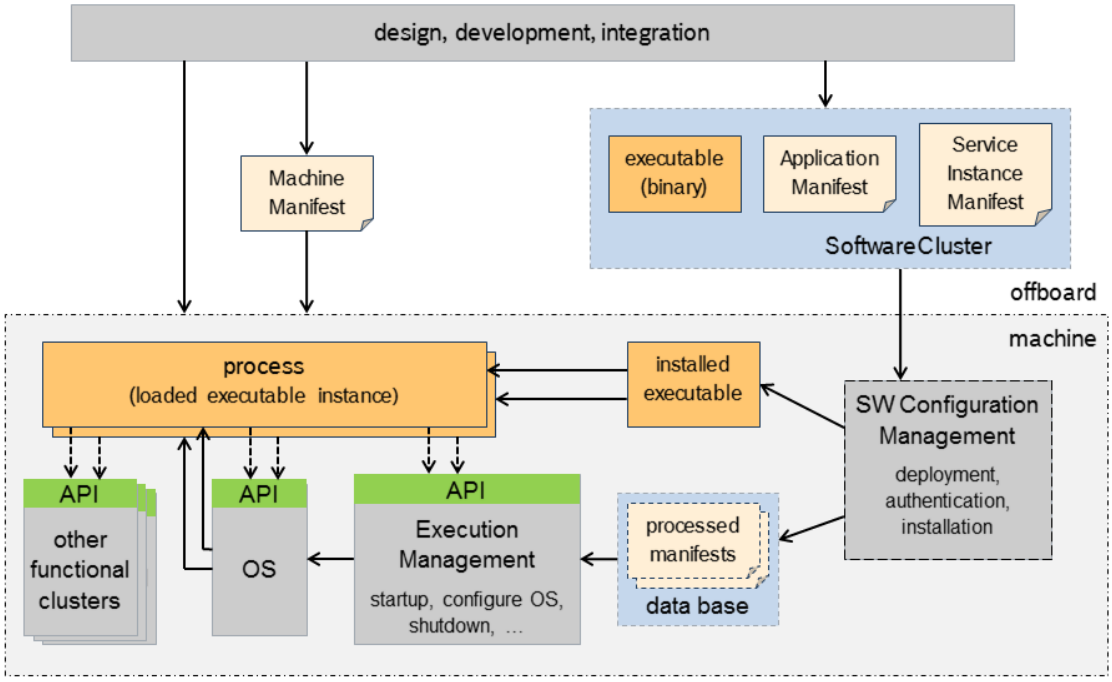

Figure 7.1: Executable Lifecycle from deployment to execution

7.2.2 Modelled Process

A Modelled Process is an instance of an Executable. On the AUTOSAR Adaptive Platform, a Modelled Process is realized at run-time as an OS process. For details on how Execution Management starts and stops Processes see Section 7.4.

Execution Management treats all Executables and the derived Modelled 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 Section 7.5).

7.2.3 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 Modelled 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 7.5 and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01017] and [TPS_MANI_01041] [2]).

To perform its necessary actions, Execution Management imposes a number of requirements on the content of the Machine Manifest and Execution Manifest. The validation of the configuration is expected to be done by the vendor tooling.

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

7.2.4 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 Modelled 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.). For details see [2].

7.2.5 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.

7.3 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.

Execution Management ensures that the integrity and authenticity of all Executables and Executable-related data (e.g. manifests) is checked. In the case of a failed integrity or authenticity check, Execution Management carries out the measures defined in Section 7.8.

[SWS_EM_02558] Default value for permissionToCreateChildProcess attribute

Upstream requirements: RS_EM_00103

⌈A Modelled Process without a specified permissionToCreateChildProcess attribute shall be considered by Execution Management as though the attribute was set to false.⌋

[SWS_EM_02559] Restriction of process creation right for processes

Upstream requirements: RS_EM_00009

⌈If permissionToCreateChildProcess attribute is false then Execution Management shall restrict the rights of the created process such that it cannot start / create other processes.⌋

The mechanism by which the restriction of [SWS_EM_02559] is implementation-specific, but could be realized by configuring the process capability attribute mask at the time of process creation (OS feature).

Please note that when setting permissionToCreateChildProcess to true, the integrator should consider several aspects:

- the safety and security implications of this configuration. Does this software come from a trusted source? Is this configuration absolutely necessary for the software to fulfill its task? etc.
- the process that creates child processes is responsible for termination of processes that it creates; processes not started by Execution Management are not controlled by Execution Management and exist outside of AUTOSAR Adaptive Platform.

- the system resources used by the child processes are generally unplanned. It is recommended to assign the parent to a dedicated ResourceGroup to limit the impact on memory and CPU usage.

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] [2].

Execution Management is **not** responsible for run-time scheduling of Processes since this is the responsibility of the Operating System [SWS_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.

Execution Management does not perform standardized termination handling - the response to receipt of a signal, e.g. SIGTERM, by Execution Management is therefore implementation defined.

7.3.1 Error handling

All API operations can potentially raise errors.

[SWS_EM_02547] Obtain error information

Upstream requirements: RS_EM_00150

⌈According to Adaptive Platform Core [9], Execution Management shall provide means to obtain information about errors that occurred during API calls. For that reason, Execution Management can return the following errors:

- ara::exec::ExecErrc [SWS_EM_02281]
- ara::exec::ExecException [SWS_EM_02282]
- ara::exec::ExecException::ExecException [SWS_EM_02283]
- ara::exec::ExecErrorDomain [SWS_EM_02284]
- ara::exec::ExecErrorDomain::ExecErrorDomain [SWS_EM_02286]
- ara::exec::ExecErrorDomain::Name [SWS_EM_02287]
- ara::exec::ExecErrorDomain::Message [SWS_EM_02288]

- `ara::exec::GetExecErrorDomain` [SWS_EM_02290]

⌋

[SWS_EM_02548] Create error information

Upstream requirements: RS_EM_00150

⌈According to Adaptive Platform Core [9], `Execution Management` shall provide means to create error information as listed below.

- `ara::exec::ExecErrorDomain::ThrowAsException` [SWS_EM_02289]
- `ara::exec::MakeErrorCode` [SWS_EM_02291]

⌋

7.4 Process Lifecycle Management

7.4.1 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 `ara::exec::ExecutionState` enumeration, see [SWS_EM_02000].

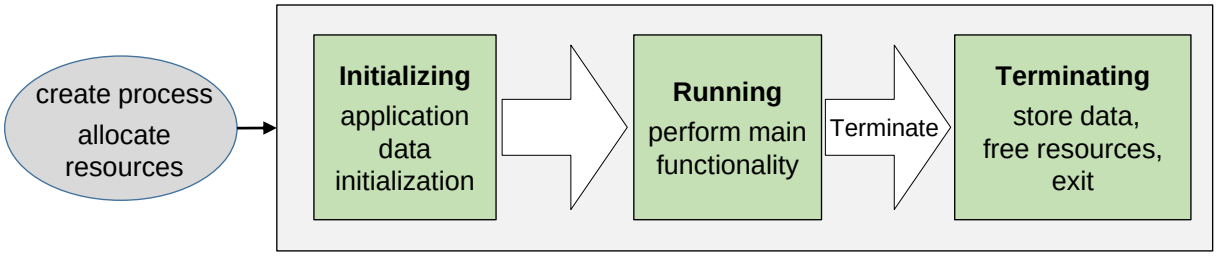

Figure 7.2: Execution States

The Execution State of a `Process` is used by `Execution Management` to construct and maintain the `Process State` as described in Section 7.4.2. Execution State change notifications from a `Process` result in `Process State` changes managed by `Execution Management`. The Execution State and `Process State` are maintained separately so that there is no explicit dependency between a `Process`'s Execution State and `Execution Management`'s `Process State`. This allows future evolution of `Process State` without impacting the internal Execution State of the `Process`.

7.4.1.1 Initialization

[SWS_EM_01401] ExecutionClient usage restriction

Upstream requirements: RS_EM_00103

⌈The `AUTOSAR Adaptive Platform` implementation shall only allow a `Process` to report its own ExecutionState.⌋

Because `ara::exec::ExecutionClient` handles the `SIGTERM` signal (see [SWS_EM_02560], [SWS_EM_02562], or [SWS_EM_02577]), it is not possible to have multiple instances of this class. Creating a second instance of `ara::exec::ExecutionClient` is unnecessary and creates ambiguity. Please consider following scenario, first instance is created with `foo()` as a termination handler and second instance is created with `bar()` as a termination handler. Which termination handler should be invoked, when `SIGTERM` is delivered?

[SWS_EM_02582] Single instance of ExecutionClient

Upstream requirements: RS_EM_00103

⌈`ara::exec::ExecutionClient` shall treat it as a Violation, when additional instance is being created.⌋

`Execution Management` considers `Process` initialization complete when the `Process State` `Running` is reached whether this is achieved implicitly (by a `Non-reporting Process`) or explicitly through a `Process` reporting its Execution State.

A `Process` is required (see [SWS_EM_01004]) to report `kRunning` state using the `ara::exec::ExecutionClient::ReportExecutionState` [SWS_EM_02003] method of class `ara::exec::ExecutionClient`, see [SWS_EM_02001]. It would typically report after the completion of its initialization, but before `Service Discovery` is completed. If the `Process` were to report `kRunning` only after `Service Discovery` completion, the non-deterministic delays may impact other `Processes`, due to delays in resolution of `Execution Dependencies`.

7.4.1.2 Termination

[SWS_EM_01055] Initiation of process termination

Upstream requirements: RS_EM_00103

⌈`Execution Management` shall initiate `Process` termination by sending the `SIGTERM` signal to the `Process`.⌋

Note that from the perspective of `Execution Management`, requirement [SWS_EM_01055] only requests the initiation of the steps necessary for graceful termination under the control of the `Process`.

It is possible that a process that should be terminated according to [SWS_EM_01055], e.g. during the handling of `Execution Dependencies`, is no longer alive. However, as `Execution Management` can determine the status of child processes it would thus not attempt to terminate a process that no longer exists.

`Execution Management` may send `SIGTERM` at any time, even before the `Process` has reported `kRunning` state and thus the `Process` is still in the `Initializing Process State`.

On receipt of `SIGTERM`, a `Process` commences the actual termination.

`Execution Management` provides a termination handler to minimize the required signal handling in the `Adaptive Application` (see [SWS_EM_02577]).

[SWS_EM_02577] Call of Termination Handler

Upstream requirements: RS_EM_00103

⌈On receipt of SIGTERM ara::exec::ExecutionClient shall invoke the termination handler defined by ara::exec::ExecutionClient::Create.⌋

Please note the API defines the execution context of the termination handler to be a background thread which is maintained by the ara::exec::ExecutionClient. It is not executed in a signal handler context, therefore usage of POSIX and ara APIs is not restricted (potential restrictions in the contract of ara APIs still apply).

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 termination with exit status 0 (EXIT_SUCCESS).

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.

[SWS_EM_01314] Default value for terminationBehavior

Upstream requirements: RS_EM_00103

⌈Execution Management shall treat a Modelled Process without specified terminationBehavior as a Process that terminates only on request by Execution Management.⌋

7.4.1.3 Unexpected Termination

[SWS_EM_01309] Unexpected Termination of a process

Upstream requirements: RS_EM_00103

⌈In case of Unexpected Termination outside a state transition resulting from previous request from ara::exec::StateClient::SetState, Execution Management shall perform the following actions:

1. log event, if logging is activated
2. Set the Function Group State (of the Function Group to which the relevant Modelled Process was mapped) to Undefined Function Group State.
3. Call *undefinedStateCallback* defined by ara::exec::StateClient with ara::exec::ExecutionErrorEvent.

⌋

If the State Management decides to invoke the ara::exec::StateClient::GetExecutionError interface, after receiving the *undefinedStateCallback* [SWS_EM_01309] or inside the callback, the ara::exec::ExecutionErrorEvent will contain the same information as provided by the *undefinedStateCallback*. This is guaranteed by [SWS_EM_02542]. Please note that [SWS_EM_01309] also applies for Unexpected Self-termination.

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

7.4.1.4 Application Reporting

[SWS_EM_02243] Handling Execution State Running

Upstream requirements: RS_EM_00103

⌈Execution Management shall return kInvalidTransition when a Process reports Execution State kRunning (using the method ara::exec::ExecutionClient::ReportExecutionState) and the Process is not in Process State Starting.⌋

To prevent denial-of-service attacks on Execution Management an implementation could rate-limit acceptance of Execution State reports or could request the Operating System to terminate the underlying process. However such reactions are not standardized.

Execution Management differentiates between two types of Processes: Reporting Processes and Non-reporting Processes. Reporting Processes are considered to be the normal form of Processes and Non-reporting Processes are considered to be an exception.

Non-reporting Processes can be used to support running Executables which have not been designed with the AUTOSAR Adaptive Platform in mind. For example, if an Executable is available as binary only, if it is not feasible to patch its source code or if the Executable is only used during development time.

The implicit transition to Running Process State is described by [SWS_EM_01402]

In safety related systems the system designer has to use Non-reporting Process functionality with care. Such Processes will probably not provide safety critical functionality and will not be monitored by Platform Health Management but still they might influence other safety related Processes and therefore can introduce a safety risk. To isolate Non-reporting Processes from safety critical parts ResourceGroup can be used (see Section 7.6).

An attempt to report *Execution State* by a Non-reporting Process is considered an error by Execution Management.

[SWS_EM_01403] Reporting Non-reporting Process

Upstream requirements: RS_EM_00103

⌈`ara::exec::ExecutionClient::ReportExecutionState` shall treat it as a Violation when invoked by a `Non-reporting Process`.⌋

7.4.2 Process States

`Process States` characterize the lifecycle of a `Process` from the point of view of `Execution Management`. In other words, `Process States` represent the `Execution Management` internal tracking of the *Execution States* (see Section 7.4.1) and hence there is no need for a standardized type. Note that each `Process` is independent and therefore has its own `Process State`. `Process State` is used by `Execution Management` to resolve Execution Dependencies, manage timeouts, etc.

Additionally to the existing values for the `Process State` (`Idle`, `Starting`, `Running`, `Terminating`, `Terminated`), the implementation may define its own `Process States`, which are not in conflict/not replacing the existing ones.

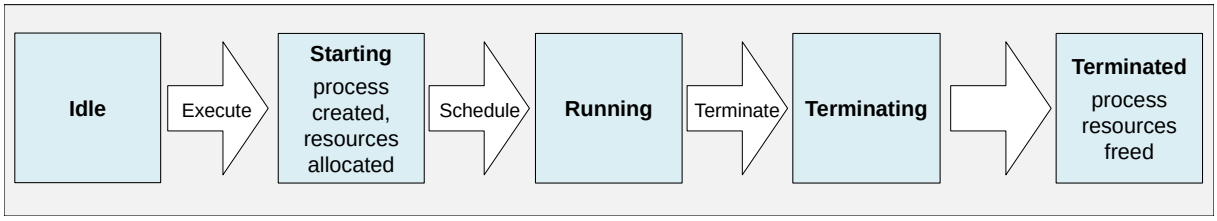

Figure 7.3: process Lifecycle

[SWS_EM_01002] Idle Process State

Upstream requirements: RS_EM_00103

⌈The `Idle` `Process State` shall be the `Process State` prior to creation of the `Process` and to resource allocation.⌋

[SWS_EM_01003] Starting Process State

Upstream requirements: RS_EM_00103

⌈The `Starting` `Process State` shall apply when the `Process` has been created and resources have been allocated.⌋

[SWS_EM_01004] Running Process State of Reporting Processes

Upstream requirements: RS_EM_00103

⌈The `Running` `Process State` shall apply to a `Reporting Process` after it has reported `kRunning` Execution State to `Execution Management`.⌋

[SWS_EM_01402] Implicit Running Process State

Upstream requirements: RS_EM_00103

⌈For Non-reporting Process the transition from Starting to Running Process State shall implicitly apply after Execution Management has allocated the required resources and created the run-time process.⌋

[SWS_EM_01404] Terminating Process State after Termination Request

Upstream requirements: RS_EM_00103, RS_EM_00011

⌈The Terminating Process State shall apply when Execution Management sent SIGTERM signal to the Process.⌋

[SWS_EM_01006] Terminated Process State

Upstream requirements: RS_EM_00103, RS_EM_00011

⌈The Terminated Process State shall apply after the Process has terminated and the Process resources have been freed.⌋

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]. The distinction between Process State Idle and Terminated is relevant for resolving Execution Dependencies to Self-terminating Processes (see Section 7.4.4.1).

7.4.2.1 Synchronization with Platform Health Management

Platform Health Management requires Process State information for starting and stopping of Supervisions. For details see [7].

Platform Health Management needs the information that a Supervised Process reported Execution State kRunning ([SWS_EM_01004]) to start Alive Supervision.

[SWS_EM_01210] Report "kRunning received event" to Platform Health Management

Upstream requirements: RS_EM_00103

⌈Execution Management shall inform Platform Health Management if a Supervised Process has reported Execution State kRunning.⌋

Platform Health Management needs the information that termination of a Supervised Process will be initiated ([SWS_EM_01055]) to stop Intra-process Supervisions.

[SWS_EM_01211] Report "initiating process termination" event to Platform Health Management

Upstream requirements: RS_EM_00103

⌈Execution Management shall inform Platform Health Management when a Supervised Process termination is about to be initiated.⌋

Platform Health Management needs the information that a Supervised Process is terminated ([SWS_EM_01006]) to supervise Self-terminating Processes.

[SWS_EM_01212] Report "process terminated" event to Platform Health Management

Upstream requirements: RS_EM_00103

⌈Execution Management shall inform Platform Health Management when a Supervised Process is terminated.⌋

Hint: Which Processes are Supervised by Platform Health Management can be determined by referring to the configuration of Platform Health Management.

The above notifications are provided through Inter-Functional Cluster Interface(s) between Execution Management and Platform Health Management. As such interfaces are vendor-specific, their definition (signature) is not standardized.

If Execution Management is used in Safety Critical Platform, then it is suggested to use Alive/Logical/Deadline supervision(s) and report their checkpoints appropriately to Platform Health Management.

7.4.3 Trace Process State Transitions

The timing behavior of Processes is a mandatory part of the lifecycle of an application. Wrong timing and timing shortages in Processes can lead to unexpected behavior or failures. Therefore it is important to provide a way to trace Process lifetime events within the Execution Management. The following trace points are introduced to be able to do a timing analysis of an application based on its Processes.

[SWS_EM_02573] State Transition logging – process created

Upstream requirements: RS_EM_00152

⌈Whenever Execution Management has created a process (see [SWS_EM_01003]), Execution Management shall log a DltMessage of type ProcessCreated.⌋

[SWS_EM_02574] State Transition logging – process kRunning received

Upstream requirements: RS_EM_00152

⌈Whenever Execution Management has received a kRunning (see [SWS_EM_02003]), Execution Management shall log a DltMessage of type ProcessKRunningReceived.⌋

[SWS_EM_02575] State Transition logging – process termination request

Upstream requirements: RS_EM_00152

⌈Whenever Execution Management is requesting a process to be terminated (see [SWS_EM_01055]), Execution Management shall log a DltMessage of type ProcessTerminationRequest.⌋

[SWS_EM_02576] State Transition logging – process terminated

Upstream requirements: RS_EM_00152

⌈Whenever a process terminates (independent of a success return result), Execution Management shall log a DltMessage of type ProcessTerminated.⌋

7.4.4 Startup and Termination

7.4.4.1 Execution Dependency

Execution Management can derive an ordering for the startup and termination of Processes within State Management framework based on the declared Execution Dependencies. 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] [2] and [constr_1606] [2], are configured in the Execution Manifests, which is created at integration time based on information provided by the application developer. 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.

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 7.5. The integrator should ensure that all service dependencies are mapped to the 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 7.4.1) 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 7.4.2.

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 achieves Running Process State.

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.

A Terminated Execution Dependency specified on a non self-terminating Process is considered to be a configuration error as this would indicate a dependency that can only be fulfilled at the next group transition [SWS_EM_CONSTR_00001]

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. See also [SWS_EM_CONSTR_01744].

[SWS_EM_CONSTR_01744] Definition of process state in the context of the Execution Dependency

Upstream requirements: RS_EM_00100

⌈The target ModeDeclaration referenced in the role ExecutionDependency.processState shall fulfill the following conditions:

- It shall be owned by a ModeDeclarationGroup that is referenced by a ModeDeclarationGroupPrototype (in the role type) that in turn shall be aggregated by a Process.
- The shortNames of the encapsulated ModeDeclarations shall only be one of the following values:
  - Running

- Terminated

⌋

[SWS_EM_CONSTR_00001] Modeling execution dependency for the `Terminated` state

Upstream requirements: RS_EM_00100

⌈A Terminated ModeDeclaration referenced in the Process.stateDependentStartupConfig.executionDependency shall only be allowed if the process referenced in the stateDependentStartupConfig.executionDependency has StartupConfig.terminationBehavior set to processIsSelfTerminating.⌋

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.

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. Note that the scope of Execution Dependency resolution is limited to one Function Group State only (see [constr_1689] [2] and [SWS_EM_02245]).

[SWS_EM_01050] Start Dependent processes

Upstream requirements: RS_EM_00100

⌈During startup of a Process, Execution Management shall respect Execution Dependencies by ensuring that each Process upon which the Process to be started depends have reached its requested Process State (at some previous point in time) during the current state transition before starting the Process.⌋

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

[SWS_EM_01051] Termination of processes

Upstream requirements: RS_EM_00100

⌈During termination of a Process, Execution Management shall respect Execution Dependencies by ensuring that each Process upon which the Process to be terminated depends is not terminated before termination of the Process.⌋

Example 7.2

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

Note that [SWS_EM_01051] merely requires Execution Management to not terminate the dependent Processes before terminating 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 terminated in an arbitrary order.

Example 7.3

Consider three Processes:

- ***Storage***, a service Process without any dependencies;
- ***StorageConsistencyChecker***, a self-terminating Process, it requires ***Storage*** to be in Process State Running;
- ***ConfigReader***, a service Process, it requires that the ***StorageConsistencyChecker*** has reached Process State Terminated;

For startup this means Execution Management should start ***Storage*** and wait till it reports kRunning, then Execution Management should start ***StorageConsistencyChecker*** and wait till it terminates and only then start ***ConfigReader***. For termination the Execution Dependency indicates that Execution Management can terminate ***Storage*** and ***ConfigReader*** simultaneously because ***StorageConsistencyChecker*** is already terminated and ***ConfigReader*** does not have a direct dependency on ***Storage***. If ***ConfigReader*** has to be terminated before ***Storage***, then this can be achieved by adding a direct Execution Dependency between ***ConfigReader*** and ***Storage***.

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].

[SWS_EM_01001] Execution Dependency error

Upstream requirements: RS_EM_00100

⌈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 consider this as an Error and fail to start Process A.⌋

Example 7.4

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 requirement [SWS_EM_01001] effectively forbids any Execution Dependencies that spans outside of a single Function Group State (or a Machine State) definition, see also [constr_1689] [2]. This is done on purpose, as this kind of dependencies will introduce hidden dependencies between Function Groups or Function Group States and they will not be visible to State Management. For an illustration on which Execution Dependencies are permitted see Figure 7.13. If dependencies between Function Groups need 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 [11].

Unlike a Reporting Process, a Non-reporting Process is in Process State Running directly after start. Regardless of whether the process has completed its initialization phase and is ready to offer its services or not. This means that Running Execution Dependencies are immediately satisfied and thus do not achieve the original semantics when specified for a Non-reporting Processes without further action.

This limitation can be overcome by introducing a Companion Process, which acts as a representative of the Non-reporting Process. The Companion Process waits for availability of the service provided by the Non-reporting Process and reports kRunning to Execution Management. The Processes which in fact need the services of the Non-reporting Process can be configured to be dependent on the Companion Process. Please note that the Terminated Execution Dependency is not affected as Execution Management is informed by the Operating System when Non-reporting Processes are Terminated. Please see Figure 7.4 for more details.

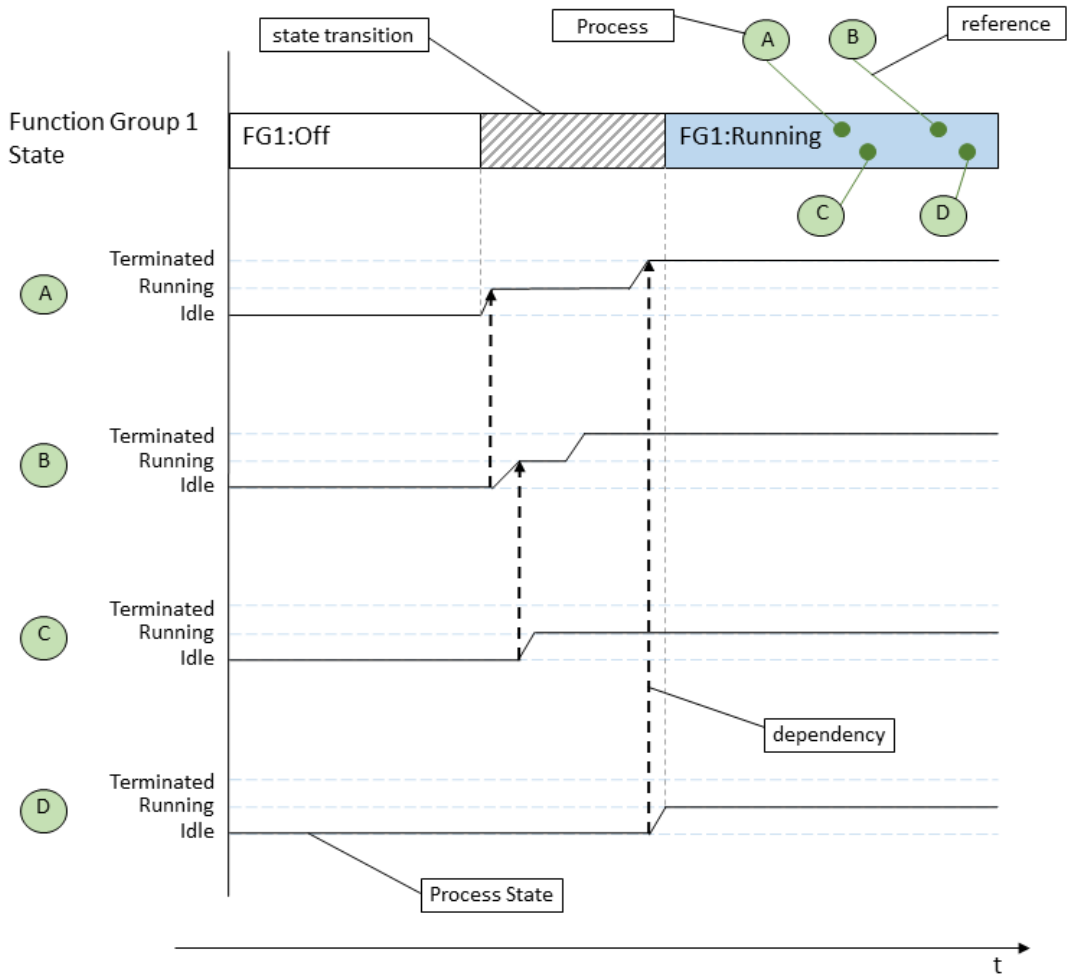

Figure 7.4: Execution dependencies on Non-reporting Process

- `Non-reporting Process` (and `Self-terminating Process`) **A** references `FG1:Running`. This process is started first (as it doesn't have any `Execution Dependencies` configured) and automatically enters **Running** `Process State` as per [SWS_EM_01402].
- `Companion Process` **B** is started after `Non-reporting Process` **A** (please note that **A** and **B** are also standard AUTOSAR `Processes`) enter **Running** state. `Process` **B** can use project specific method to assess if `Process` **A** is fully functional and signal this to `Execution Management` by reporting (or not) `kRunning` state.
- `Process` **C** is started when (and only when) `Process` **B** enters **Running** `Process State` (i.e. reports `kRunning`). Please note this `Execution Dependency` will work independently from reporting / non-reporting configuration of `Process` **C**.

- Process **D** has **Terminated** Execution Dependency configured on Self-terminating Process (and Non-reporting Process) **A**. As mentioned earlier this works out of the box (no special action needed here).

7.4.4.2 Signal Mask

AUTOSAR Adaptive Platform is defining a POSIX based system for applications - this includes POSIX signals. Signal handling is not a trivial task. Therefore interaction with POSIX signals is limited to a necessary minimum in AUTOSAR Adaptive Platform. The only reason for POSIX signal interaction is Execution Management's Process termination request (see [SWS_EM_01055]).

Execution Management starts Processes with a well defined signal mask to avoid the situation of threads having an undefined signal mask (e.g. threads created by ara).

[SWS_EM_02578] Initial signal mask for Reporting Process

Upstream requirements: RS_EM_00103

⌈Execution Management shall start Reporting Processes with a signal mask having all POSIX signals blocked except of:

- SIGABRT
- SIGBUS
- SIGFPE
- SIGILL
- SIGSEGV

⌋

The set of unblocked POSIX signals provided in [SWS_EM_02578] are raised by the OS when an error occurs where there is no reasonable way to continue program execution. According to POSIX, ignoring these signals would lead to undefined behavior. If one of these signals occurs, the default signal handling action, in this case abnormal termination, is performed. Note also that the two POSIX signals SIGSTOP and SIGKILL cannot be caught or blocked.

Adaptive Application make use of ara::exec::ExecutionClient which implements catching the SIGTERM signal (see [SWS_EM_02577]).

Non-reporting Processes won't use the ara::exec::ExecutionClient and will implement their own signal handling. For such applications Execution Management prepares an empty signal mask.

[SWS_EM_02579] Initial signal mask for Non-Reporting process

Upstream requirements: RS_EM_00103

⌈`Execution Management` shall start `Non-reporting Processes` with an empty signal mask.⌋

7.4.4.3 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] Process Argument Passing

Upstream requirements: RS_EM_00010

⌈At the initiation of startup of a `Process`, the aggregated `ProcessArgument` of the `StartupConfig` referenced by the `StateDependentStartupConfig` shall be passed to the `Process` by `Execution Management` based on [SWS_EM_01072] and [SWS_EM_01078].⌋

Note that [SWS_EM_01012] deliberately does not specify the OS mechanism used to start a `Process`, e.g. the `exec`-family based POSIX interface, as this is ultimately an implementation specific property.

The first argument passed by `Execution Management` is the name of the `Executable`.

[SWS_EM_01072] process Argument Zero

Upstream requirements: RS_EM_00010

⌈Argument 0 shall be set to name of the `Executable`.⌋

`Execution Management` supports passing arguments to a `Process` in the same way that a shell passes command line arguments to a POSIX process. `Execution Management` assigns each `argument.argument` to an element in the `argv[]` array, starting at element index 1, and passes this to the `Process` `main()` function. `ProcessArgument` ordering is used to preserve the semantics of an `(option, argument)` pair such as "`-b value`", where the "`-b`" argument must precede the "`value`" argument. This method supports the short form and long form argument passing conventions typically used in POSIX environments.

[SWS_EM_01078] Process Argument strings

Upstream requirements: RS_EM_00010

⌈`ProcessArgument.argument` shall be passed to the `Process` in order with the first `ProcessArgument.argument` starting at Process Argument 1.⌋

The order in which the defined `ProcessArgument` are passed is defined by the ordered `StartupConfig.processArgument` aggregation.

7.4.4.4 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] process specific Environment Variables

Upstream requirements: RS_EM_00010, RS_AP_00130

⌈`Execution Management` shall prepare environment variables based on the configuration from `Process.stateDependentStartupConfig.startupConfig.environmentVariable` and pass them during a `Process` start.⌋

[SWS_EM_02247] Machine specific Environment Variables

Upstream requirements: RS_EM_00010, RS_AP_00130

⌈`Execution Management` shall prepare environment variables based on the configuration from `Machine.environmentVariable` and pass them during a `Process` start.⌋

Please note that AUTOSAR meta model uses `TagWithOptionalValue` for environment variables definition ([TPS_MANI_01208] and [TPS_MANI_01209] [2]). As explained there, the value (`TagWithOptionalValue.value`) can be omitted as a way of specifying environment variable with empty value.

[SWS_EM_02249] Missing value from Environment Variable definition

Upstream requirements: RS_EM_00010, RS_AP_00130

⌈Whenever `Execution Management` finds environment variable definition, that has `TagWithOptionalValue.value` missing, it should use empty string as a value for this environment variable.⌋

[SWS_EM_02248] Environment Variables precedence

Upstream requirements: RS_EM_00010, RS_AP_00130

⌈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.⌋

7.4.5 Machine Startup Sequence

Execution Management is the AUTOSAR Adaptive Platform's first process. When ready, Execution Management initiates the Machine State transition from the Off state (the default state before EM is started) to the Startup state ([SWS_EM_01023], [SWS_EM_02250]). During the transition, Execution Management requests startup of processes that exist in the Startup Machine State.

After the necessary state transition conditions have been met (see Section 7.5.5 and Section 7.5.2.1), Execution Management reports Machine State Startup transition confirmation to State Management ([SWS_EM_02241]). At that point, Execution Management hands over responsibility for Function Group state management (i.e. initiation of state change requests) to State Management.

On a Machine, which can be any group of resources, i.e. a physical environment, a virtualized environment over a hypervisor, or an OS-level virtualization (container), Execution Management is not necessarily the first process launched; Other processes needed by the system may exist, such as an Operating System init process, or an Operating System Micro-kernel user-level processes like drivers, filesystem, etc. All of these processes might be started and managed outside of the context of the AUTOSAR Adaptive Platform.

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] Startup order

Upstream requirements: RS_EM_00100

⌈The startup order of the platform-level Processes shall be determined by Execution Management based on Machine Manifest and Execution Manifest information.⌋

Please see Section 7.2.3.

Figure 7.5 shows the overall startup sequence.

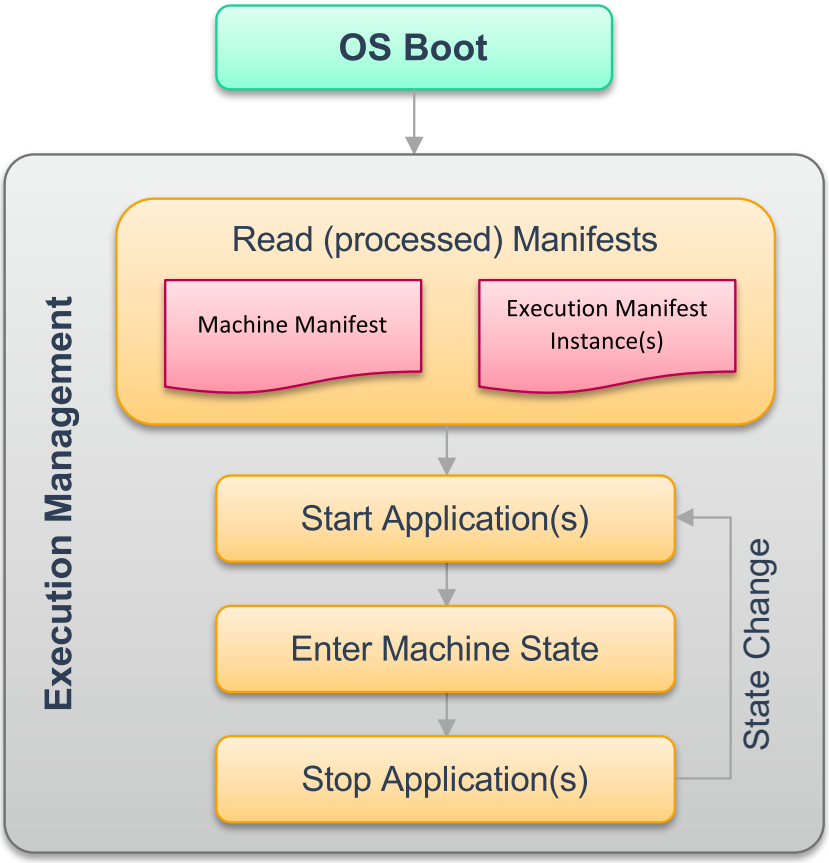

Figure 7.5: Startup sequence

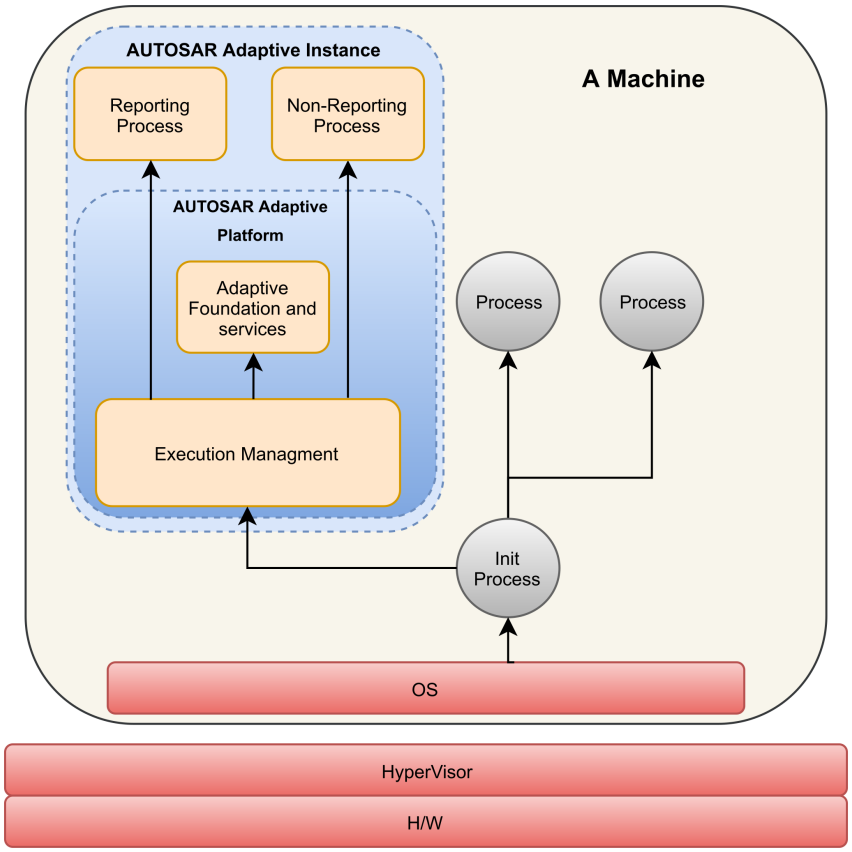

Figure 7.6: AUTOSAR Adaptive Platform Boundary

7.5 State Management

7.5.1 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 Modelled Processes have to run (see [2]). As mentioned before, a Modelled Process is an instance of an Executable, which is part of an Adaptive 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 7.4.1

Process State – Process States are managed by an Execution Management internal state machine. For details see Section 7.4.2.

Machine State – see Section 7.5.2

Function Group State – see Section 7.5.3

An example for the interaction between these states will be shown in section Section 7.5.4.

7.5.2 Machine State

[SWS_EM_CONSTR_02556] Mandatory states

Upstream requirements: RS_EM_00101

⌈Execution Management requires that exactly one Function Group with the name "MachineFG" is configured for each Machine. This Function Group has several mandatory states:

- **Off**,
- **Verify**,
- **Startup**,
- **Shutdown**, and
- **Restart**.

⌋

In order to clarify the required contents of the "MachineFG" function group and its type, the ARXML model AUTOSAR_MOD_GeneralDefinition_MachineFG.arxml (available in MOD_GeneralDefinitions) provides a definition.

Additional Machine States can be defined on a machine specific basis and are therefore not standardized.

The Execution Manifest defines the relation between processes and Function Group States. Therefore it is possible to determine the set of executed processes for each Function Group State. A Function Group State is modeled by means of ModeDeclaration, see [TPS_MANI_03145] and [TPS_MANI_03194] [2].

In the API, a Function Group is represented by the class ara::exec::FunctionGroup, see [SWS_EM_02263] and a Function Group State by the class ara::exec::FunctionGroupState, see [SWS_EM_02269]. Class ara::exec::StateClient performs state management during the lifetime of a Machine, see [SWS_EM_02275].

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 7.5.5.

[SWS_EM_01032] Machine States configuration

Upstream requirements: RS_EM_00101

⌈Execution Management shall obtain the configuration of Machine States from Function Group "MachineFG" within the SoftwareCluster with category PLATFORM_CORE.⌋

[SWS_EM_CONSTR_02557] Scope of machine Function Group

Upstream requirements: RS_EM_00101

⌈The function-Group that represents the Machine Function Group group (see [SWS_EM_CONSTR_02556]) shall only be referenced in the role claimedFunctionGroup by a SoftwareCluster of category PLATFORM_CORE.⌋

Please note that according to [constr_1788] [2] there must be exactly one SoftwareCluster with category PLATFORM_CORE on each machine.

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 7.7.

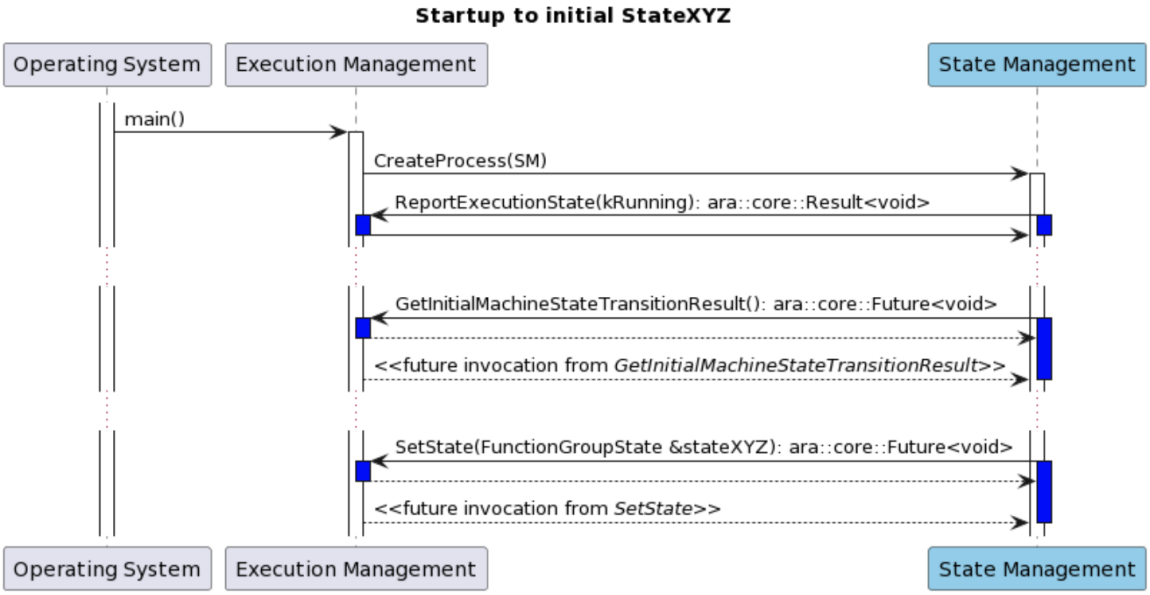

Figure 7.7: Start-up Sequence – from `Startup` to initial running state `StateXYZ`

A successful Function Group state change sequence is illustrated in Figure 7.8. 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`.

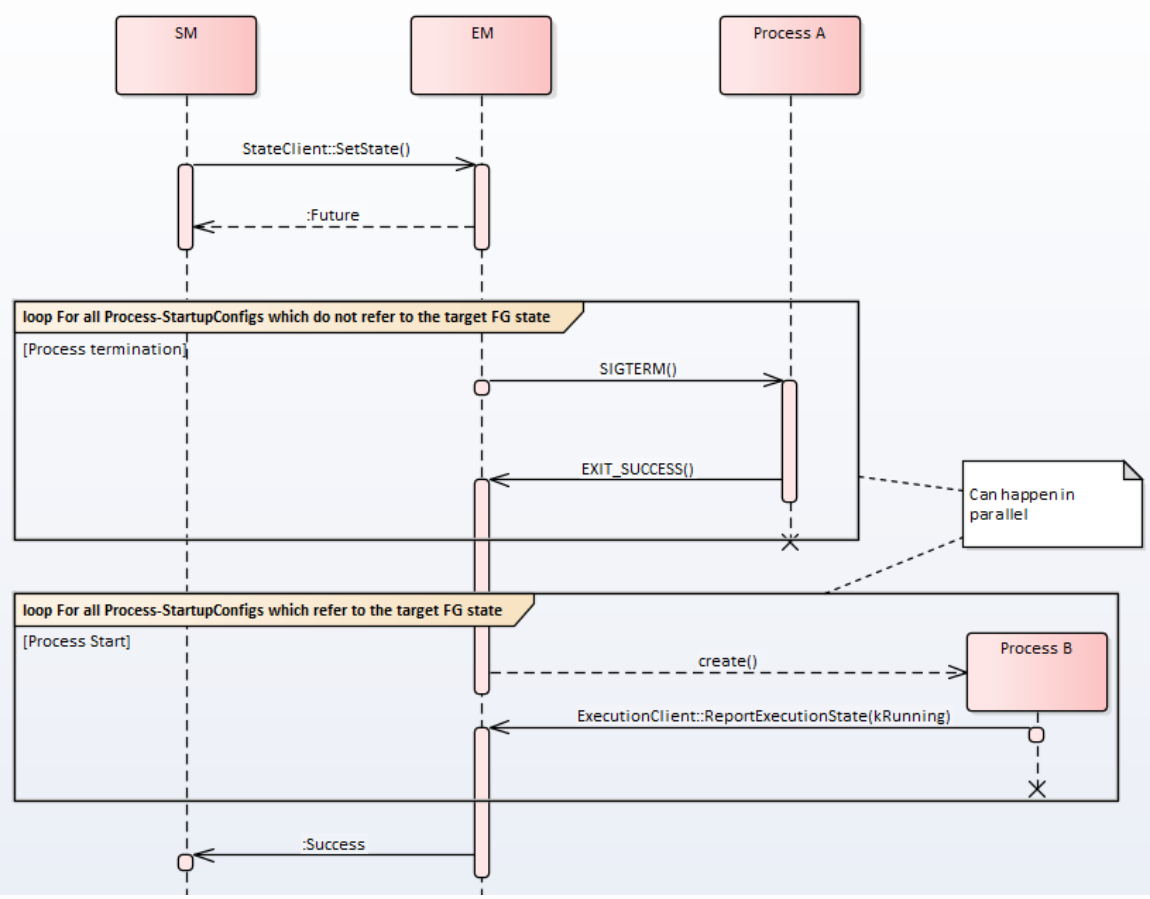

Figure 7.8: State Change Sequence - Success

Figure 7.9 shows a Function Group state change which is continued even if a termination timeout happens. Figure 7.10 shows a Function Group state change which is aborted because of a start-up timeout.

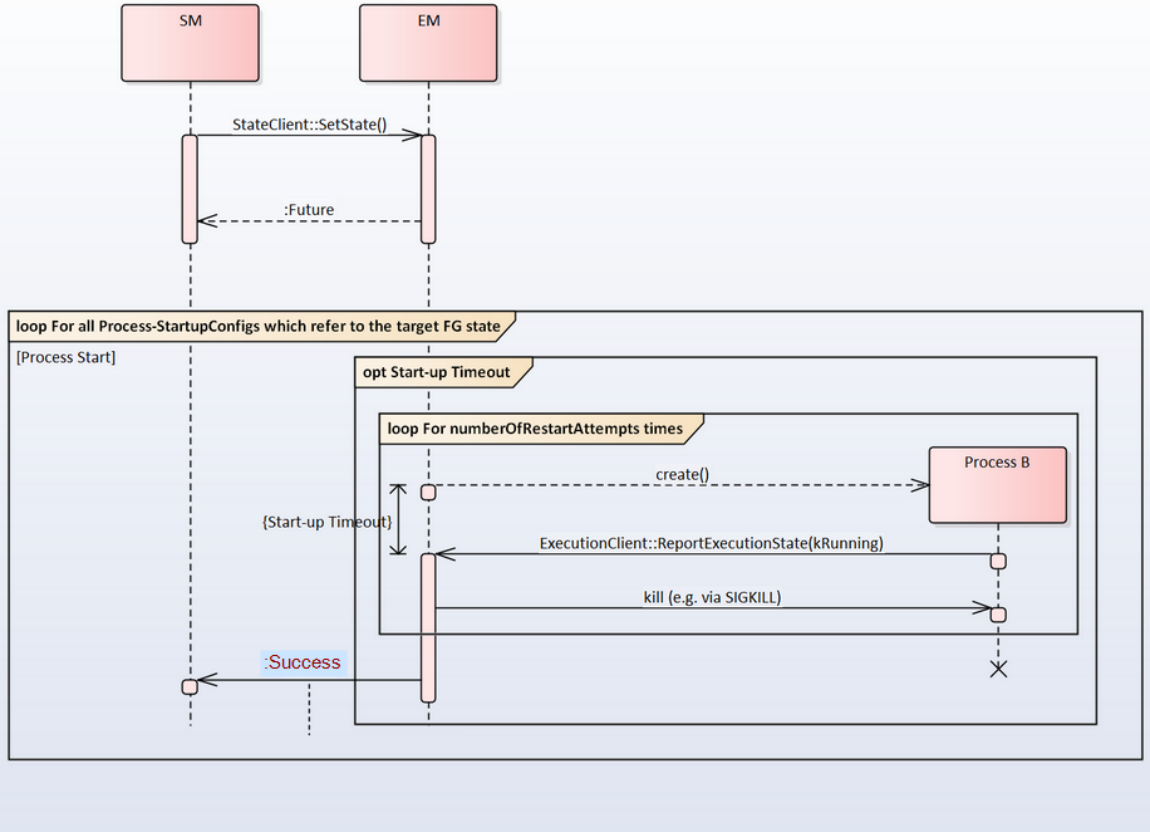

Figure 7.9: State Change Sequence - Termination Timeout

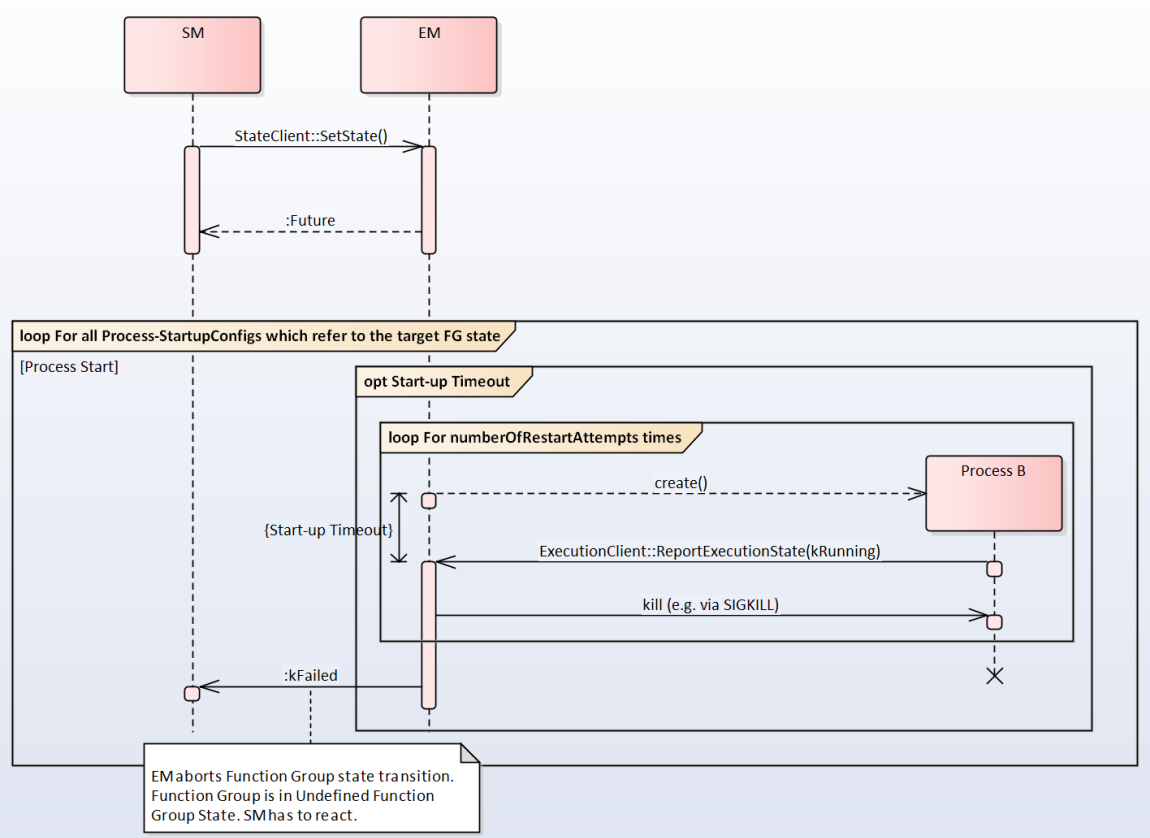

Figure 7.10: State Change Sequence - Start-up Timeout

There are more cases which lead to failure during Function Group state change, which are not illustrated here. For details see Section 7.5.5 "State Transition" and Section 8.2.4.8 "StateClient::SetState".

7.5.2.1 Startup

[SWS_EM_02250] Machine State Startup

Upstream requirements: RS_EM_00101

⌈`Execution Management` shall enter `Unrecoverable State` if `Execution Management` is not able to parse the `Startup` state information of `Function Group` "MachineFG".⌋

`Execution Management` depends on the existence of a `Startup` state [SWS_EM_CONSTR_02556]. However, at run time the state may not be available to `Execution Management` for a number of reasons (e.g. misconfigured development system, corrupted file system, memory errors etc.). While these situations may not be common, to avoid implementation specific behavior, `Execution Management` should have standardized reaction to them.

[SWS_EM_01023] Self initiation of Machine State Startup transition

Upstream requirements: RS_EM_00101

⌈Execution Management shall self initiate the state transition to the Startup Machine State.⌋

Please note that for Machine State transitions, the requirements of section Section 7.5.5 apply.

[SWS_EM_02555] Failure in Machine State Startup transition

Upstream requirements: RS_EM_00101

⌈Execution Management shall enter Unrecoverable State in the event of failed transition to the Startup Machine State.⌋

A failure in transition to Startup Machine State is considered as a serious problem. In that event Execution Management can't be sure what level of functionality is available and if a failed state transition can be handled by State Management. It is worth to note that the State Management itself can be unavailable or its functionality can be very limited at that point in time.

[SWS_EM_02241] Machine State Startup Completion

Upstream requirements: RS_EM_00101

⌈Upon completion of initial (self initiated) Machine State transition to the Startup state, Execution Management shall make the result of that transition available to State Management through ara::exec::StateClient::GetInitialMachineStateTransitionResult API.⌋

[SWS_EM_02583] Machine State Startup transition result access

Upstream requirements: RS_EM_00101

⌈ara::exec::StateClient::GetInitialMachineStateTransitionResult shall return kInvalidMetaModelIdentifier if the calling Process has no FunctionGroupPortMapping to Function Group "MachineFG".⌋

Please note that the notification in [SWS_EM_02241] is not done via broadcast message but has to be requested by State Management via the ara::exec::StateClient::GetInitialMachineStateTransitionResult API.

The function ara::exec::StateClient::GetInitialMachineStateTransitionResult retrieves the result of the Machine State's initial transition to the Startup state. After the Startup state is reached (as described by [SWS_EM_02241]) Execution Management does not initiate any further Function Group State changes (this includes Machine State). Instead such changes are requested by State Management and then performed by Execution Management.

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) [SWS_SM_-CONSTR_00001] [11]. 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.

The possibility that the Machine State transition to the Startup state is never reached shall be taken into account. In this case the State Management can interrupt the Startup state transition and request e.g. a recovery state using the ara::exec::StateClient::SetState interface. The ara::exec::StateClient::GetInitialMachineStateTransitionResult would return the value kOperationCanceled.

7.5.2.2 Shutdown/Restart

Execution Management does not perform shutdown/restart of the Machine to avoid embedding project-specific behavior within Execution Management. Instead a project-specific actor is expected to provide a mechanism to shutdown/restart the Machine, such as, a standalone process that is configured to be started by Execution Management during transition to the Shutdown / Restart Machine State or a process started in Startup Machine State that waits for a signal before shutting down the Machine. This approach enables the control of both WHEN and HOW shutdown/restart occurs to be managed in a project-specific manner. See [constr_-1618] and [constr_1619] [2].

Requirements [SWS_EM_02241] and [SWS_EM_01023] dictate a dependency by Execution Management on the presence of the Startup Machine State and [SWS_EM_CONSTR_02556] mandates configuration of Startup and Shutdown / Restart Machine States. However there is no equivalent requirement on Shutdown or Restart Machine States as their omission does not prevent Execution Management from starting. Therefore, the response by Execution Management to this misconfiguration is implementation-specific.

A request to Execution Management to change the current Machine State to either Shutdown or Restart 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 "MachineFG" Shutdown or Restart when it's valid to do so; see [11] for advice on how to orchestrate shutdown of the Machine.

Please note it is system integrator's responsibility to carefully consider when system shutdown / restart should be requested because all Processes which are still running will not be terminated by Execution Management, which means that they will not be able to persist their data.

As mentioned in Section 7.5.2.1, AUTOSAR assumes that State Management will be configured to run in Shutdown and Restart. 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 to report errors and wait for further instructions. Please note that the purpose of entering the Shutdown or Restart state is to shutdown or restart the Machine (this includes State Management) in a clean manner.

[SWS_EM_CONSTR_02558] Ability to shut down

Upstream requirements: RS_EM_00101

⌈In the context of one Machine, at least one Process shall have a stateDependentStartupConfig.functionGroupState that has the shortName Shutdown.⌋

[SWS_EM_CONSTR_02559] Ability to restart

Upstream requirements: RS_EM_00101

⌈In the context of one Machine, at least one Process shall have a stateDependentStartupConfig.functionGroupState that has the shortName Restart.⌋

[SWS_EM_02549] MachineFG.Off handling

Upstream requirements: RS_EM_00101

⌈Execution Management shall refuse a request to change "MachineFG" Function Group State to Off with error kInvalidTransition.⌋

7.5.3 Function Group State

If there is a group of applications installed on the machine, it will be useful to have ability of controlling them coherently. 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.

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 7.5.5.

A Modelled Process may not be assigned to more than one Function Group [constr_1688] [2]. To see why this constraint is required consider the contrary a Modelled Process mapped to two states in two Function Groups. The Modelled Process is now running in the two states and a Function Group State transition in either state would require the Process to be terminated. This termination would violate the integrity of the second Function Group State and hence the constraint exists to prevent this situation.

In general, Machine States (see Section 7.5.2) 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 user-level applications. Please note that this doesn't mean that all Processes of platform level applications have to be controlled by Machine States.

Figure 7.11 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.

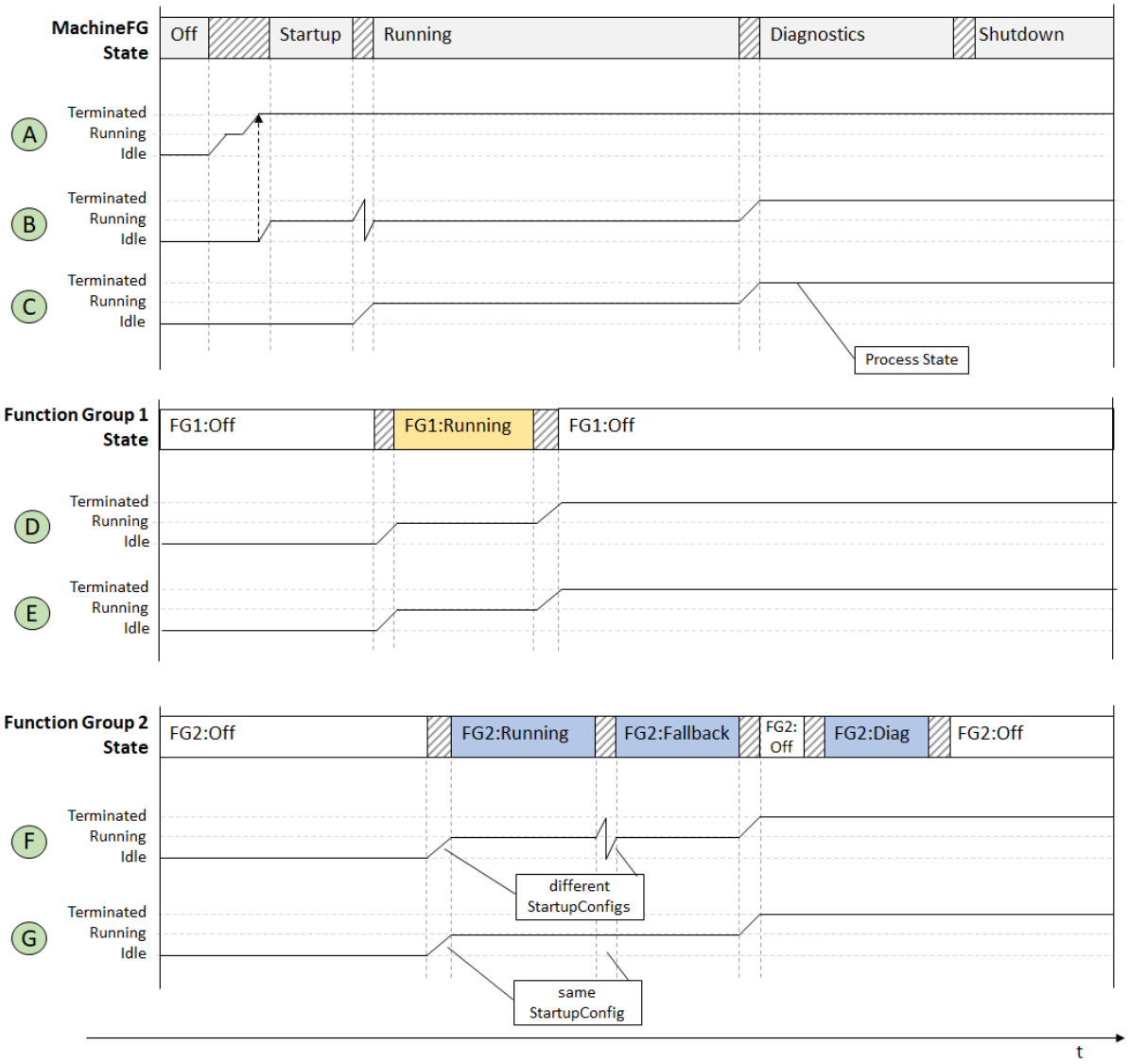

Figure 7.11: State dependent process control

- `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` from different `StateDependentStartupConfigs` [constr_10411]. Multiple startup configurations are required as the process depends on the termination of `Process` **A**, i.e. an `Execution Dependency` has been configured, as described in Section 7.4.4.1
- `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.
- Process **G** references FG2:Running and FG2:Fallback similarly to Process **F** however the states are referenced from the same StateDependentStartupConfig therefore it is not restarted at the state transition and continues execution.

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.

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. If a Process doesn't reference any Function Group States it will never be started, for more details please refer to [SWS_EM_01066] and Section 7.5.5 State Transition.

Each Modelled 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 [2]. By parsing this information from the Execution Manifests, Execution Management can determine which Modelled Processes need to be launched if a specific Function Group State is entered, and which startup parameters are valid.

[SWS_EM_01033] process start-up configuration

Upstream requirements: RS_EM_00009, RS_EM_00101

⌈To enable a Modelled Process to be launched in multiple Function Group States, Execution Management shall be able to configure the Process started on every Function Group State change based on information provided in the Execution Manifest.⌋

Please note AUTOSAR doesn't support the possibility of assigning a single Process to more than one Function Group, see [constr_1688] [2].

[SWS_EM_01110] Off States

Upstream requirements: RS_EM_00101

⌈Each Function Group (including the Function Group "MachineFG") has an Off State which shall be used by Execution Management as initial Function Group State.⌋

Within any FunctionGroup, including "MachineFG", the "Off" state is mandatory as the initial state [TPS_MANI_03195] [2] and cannot have Modelled Processes

mapped according to [constr_3424] [2]. [SWS_EM_01110] and [SWS_EM_01023] together define the very first Function Group state transition after the power up.

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 [2], especially "State-dependent Startup Configuration" chapter and "Function Groups" chapter.

The arbitrary state change sequence as shown in Figure 7.8 applies to state changes of any Function Group - just replace "MachineFG" 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 7.5.5.

7.5.4 State Interaction

Figure 7.12 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.

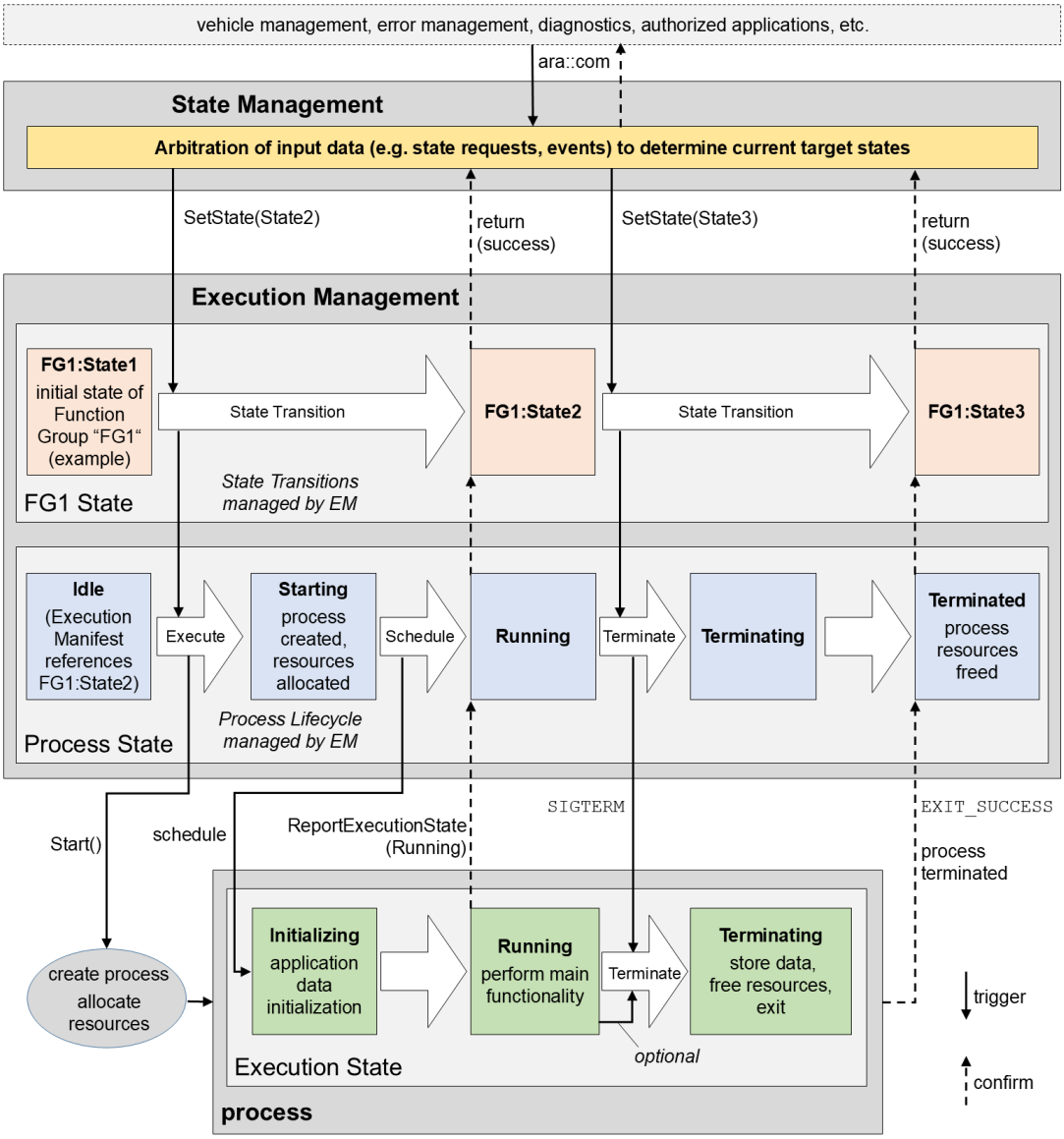

Figure 7.12: Interaction between states

7.5.5 State Transition

State Management can request to change one or several Function Group States (including the Machine State), using API described in Section 8.2.4. ara::exec::StateClient::SetState allows State Management to request several Function Group State changes in parallel. If Machine State change is required, ara::exec::FunctionGroup has to be created using an ara::core::InstanceSpecifier which is mapped to the ModeDeclarationGroupPrototype representing MachineFG.

[SWS_EM_02298] Request of a state transition different to the state that the Function Group is already in transition to

Upstream requirements: RS_EM_00101

⌈After successful validation of a `ara::exec::StateClient::SetState` call for a `Function Group` that is already under state transition, `Execution Management` shall cancel the ongoing `Function Group State` transition (and set that request's `ara::core::Future` to `kOperationCanceled`) before starting the new `Function Group State` transition (and returning a new `ara::core::Future` for the new request).⌋

[SWS_EM_02296] Request of a state transition to a state that the Function Group is already in transition to

Upstream requirements: RS_EM_00101

⌈After successful validation of a `ara::exec::StateClient::SetState` call for a `Function Group` that is already under state transition, `Execution Management` shall:

- set the `ara::core::Future` of the previous request to `kOperationCanceled` if there is one
- return a new `ara::core::Future` for the new request
- continue the ongoing `Function Group` state transition

⌋

There is only one situation where a transition to a state could be performed without a previous request. When `Execution Management` is performing the initial transition to `"MachineFG"` startup [SWS_EM_01023], State Management can decide to request a different state of `"MachineFG"`. In this situation there is no previous request, so there is no `ara::core::Future`, that can be set to `kOperationCanceled`.

Before `Execution Management` cancels an ongoing request according to [SWS_EM_02298] or [SWS_EM_02296] the new request should be assessed as valid, this includes, but is not limited to, [SWS_EM_02549].

Please note that [SWS_EM_02298] merely ensures that `Execution Management` first informs the requester of the ongoing transition (instance of `ara::exec::StateClient`) about the cancellation, before informing the new requester that the new request has been accepted. Both requesters could be the same instance of `ara::exec::StateClient`.

[SWS_EM_02295] Request of a state transition to a state that the Function Group is already in

Upstream requirements: RS_EM_00101

⌈`ara::exec::StateClient::SetState` shall ignore the request and return immediately with success.⌋

Note that an error domain value is not used for [SWS_EM_02295] as this would be interpreted as an error by State Management and thus trigger the associated error handling.

There are no other requirements or assumtions on order in which requests from `ara::exec::StateClient::SetState` are processed.

Requesting the same `Function Group State` like before (independently if the previous state request is already finished or still ongoing) shall be prevented, because it might lead to unwanted execution dependencies. When the same `Function Group State` is to be requested again another state has to be requested before. Please note that `State Management` can repeat state transition request (to the same state) if previous transition ended with error. This is allowed because a failed state transition is considered as invalid `Function Group State`.

Since `Execution Management` allows a new `ara::exec::StateClient::SetState` call to interrupt an ongoing transition and thus change the destination `Function Group State` of the transition, it may happen (especially in misconfigured system, or during the development phase) that some of `ara::exec::StateClient::SetState` requests will be issued by mistake. It is in the best interest of `Execution Management` to inform requester (instance of `ara::exec::StateClient`) of the ongoing transition, that it had been canceled by a newer request as soon as possible.

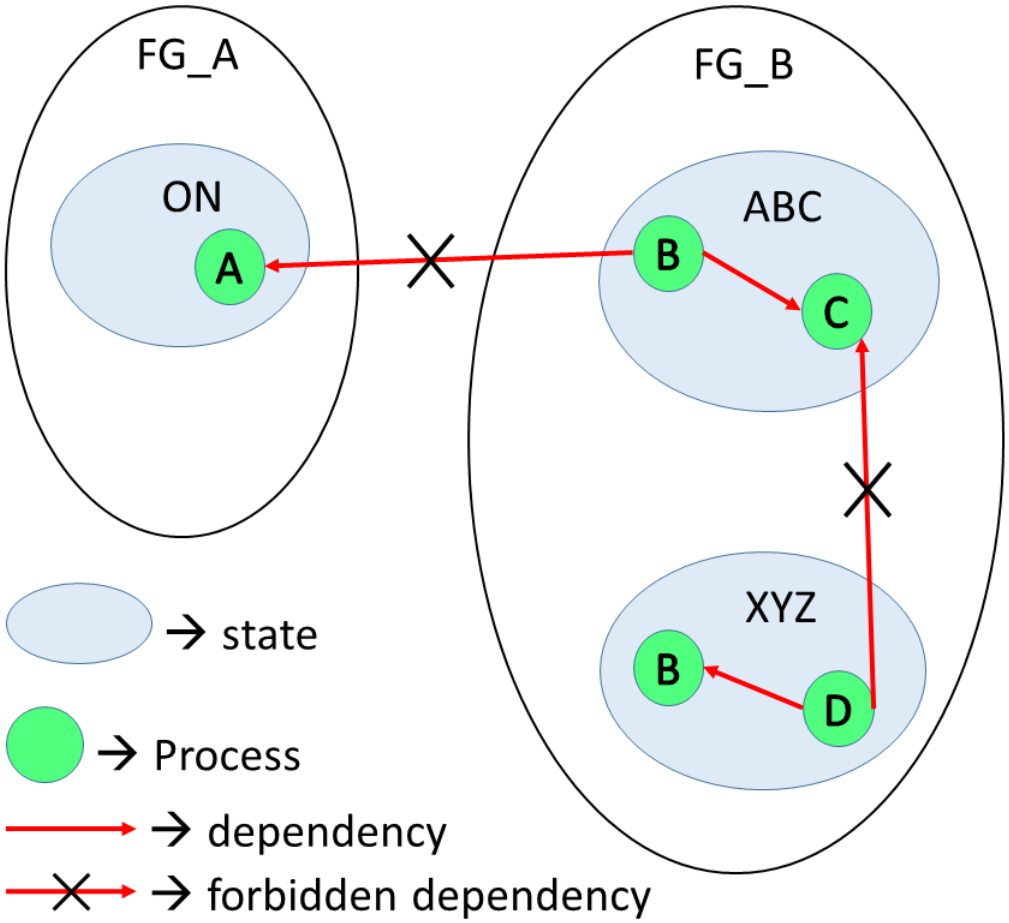

Figure 7.13: Example configuration for state transition

Before we specify how internals of a state transition works, let's consider an example configuration illustrated in figure Figure 7.13. 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 [11] 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 Section 7.4.4.1 ([SWS_EM_01001] and [constr_1689] [2]).

Please note that `Process` **B** has different `Execution Dependencies` in `Function Group State` ABC and `Function Group State` XYZ. This configuration requires existence of two different startup configurations (`StateDependentStartupConfig`), which in turns will mandate `Process` **B** restart if `State Management` request `Function Group State` change from ABC to XYZ. This is enforced by [SWS_EM_02251].

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 [11] if you need more information on this).

In the following requirements the Execution Manifest of a Modelled Process is the formal modelling of Process startup behaviour and is implemented by means of the aggregation of meta-class StateDependentStartupConfig in the role Process ([TPS_MANI_01012] [2]).

The term "the Process references a State" indicates a functionGroupState that references an instance of StateDependentStartupConfig within the StartupConfig that is applicable for the Process associated with the specific Function Group 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 "MachineFg" 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 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 7.4.1 and Section 7.4.2 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] State transition - termination behavior

Upstream requirements: RS_EM_00101

⌈On state transition Execution Management shall request termination ([SWS_EM_01055]) of each Process that references the CurrentState in its Execution Manifest, but does not reference the RequestedState and has a Process State different than [Idle or Terminated].⌋

[SWS_EM_02251] State transition - restart behavior

Upstream requirements: RS_EM_00101

⌈On state transition Execution Management shall terminate all Processes that reference the CurrentState in its Execution Manifest, but references the RequestedState with different StartupConfig and have Process State different than [Idle or Terminated].⌋

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] [2]. This value may be overwritten in the startup configuration of individual Processes by defining the termination timeout parameter in the Execution Manifest, see [TPS_MANI_01278] [2].

[SWS_EM_01065] State transition - process termination timeout monitoring

Upstream requirements: RS_EM_00101

⌈Execution Management shall monitor the time required by the Process to terminate (the time needed by the Process to reach the Terminated Process State).⌋

[SWS_EM_02255] State transition - process termination timeout reaction

Upstream requirements: RS_EM_00101

⌈In the event of a Process termination timeout (defined by configuration StartupConfig.timeout), Execution Management shall request the Operating System to forcibly terminate the underlying process.⌋

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

[SWS_EM_02258] State transition - process termination timeout reporting

Upstream requirements: RS_EM_00101

⌈When the termination of a Process resulted in the timeout, Execution Management shall log the event, if logging is activated.⌋

Execution Management continues a state-transition even in the presence of non-terminating processes, since the target Function Group State will be reached as these processes will be killed (see [SWS_EM_02255] and [SWS_EM_01060]). Continuing in case of a timeout on termination assures in particular, that the Function Group State "Off" can always be reached (provided that a process termination on OS level is always successful).

This is different in case of processes that timeout during start-up (see [SWS_EM_02259]): these processes cannot be forced to start and the Function Group State will not be reached.

[SWS_EM_01066] State transition - start behavior

Upstream requirements: RS_EM_00101

⌈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].⌋

Execution Management monitors the time required by each Process to start. The start-up timeout is defined per Process startup configuration by the system integrator in the Execution Manifest, see [TPS_MANI_01277] [2].

[SWS_EM_02253] State transition - process start-up timeout monitoring

Upstream requirements: RS_EM_00101

⌈Execution Management shall monitor the time required by the Process to start-up (the time between Execution Management requesting process creation from the operating system and the Process successfully reporting the Running Process State).⌋

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_01277] [2]. Please note that startup time for Non-reporting Processes is zero because Non-reporting Processes immediately switch from Process State Idle to Running skipping the Starting state.

[SWS_EM_02260] State transition - process start-up timeout reaction

Upstream requirements: RS_EM_00101

⌈In the event of a Process start-up timeout (defined by configuration StartupConfig.timeout), Execution Management shall attempt to restart the Process up to numberOfRestartAttempts times.⌋

Process start-up timeout is caused by a malfunction and therefore Execution Management requests termination of the Process by the operating system (e.g. using

SIGKILL) rather than requesting termination through SIGTERM as the Process is assumed to be in an erroneous state.

[SWS_EM_02280] Effect on Execution Dependency

Upstream requirements: RS_EM_00101

⌈A restart attempt according to [SWS_EM_02260] shall not fulfill any terminated dependencies.⌋

[SWS_EM_02310] State transition - process termination after start-up timeout reaction

Upstream requirements: RS_EM_00101

⌈In case a Process start-up timeout occurred after Execution Management attemted to restart the Process numberOfRestartAttempts times, Execution Management shall request the Operating System to terminate the underlying Process.⌋

[SWS_EM_02259] State transition - process start-up timeout reporting

Upstream requirements: RS_EM_00101

⌈When the start-up of a Process resulted in the timeout, Execution Management shall perform following actions:

1. Stop the Function Group State transition, so State Management can decide how to proceed.
2. log event, if logging is activated
3. Set the CurrentState to Undefined Function Group State.
4. Report kOperationFailed in the ara::exec::StateClient::SetState interface to indicate that the State change request cannot be fulfilled.
5. Report the configured executionError via the ara::exec::StateClient::GetExecutionError interface.

⌋

[SWS_EM_02552] State transition - integrity or authenticity check failed

Upstream requirements: RS_EM_00101

⌈When the start-up of a Process results in the failure of an integrity or authenticity check and strictMode is active ([SWS_EM_02305]), Execution Management shall perform following actions:

1. Stop the Function Group State transition, so State Management can decide how to proceed.
2. log event, if logging is activated

3. Set the CurrentState to Undefined Function Group State.
4. Report kIntegrityOrAuthenticityCheckFailed in the ara::exec::StateClient::SetState interface to indicate that the State change request cannot be fulfilled.
5. Report the configured executionError via the ara::exec::StateClient::GetExecutionError interface.

⌋

[SWS_EM_02312] Order of process start-up timeout reaction

Upstream requirements: RS_EM_00101

⌈Execution Management shall perform the terminate reaction [SWS_EM_02310] before reporting to State Management [SWS_EM_02259].⌋

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. Another example is a configuration where Process D depends on a Self-terminating Process C to be in Process State Terminated. Process C has to be started and terminated in the requested Function Group State to fulfill D's Execution Dependency. Please see Figure 7.14 for more details.

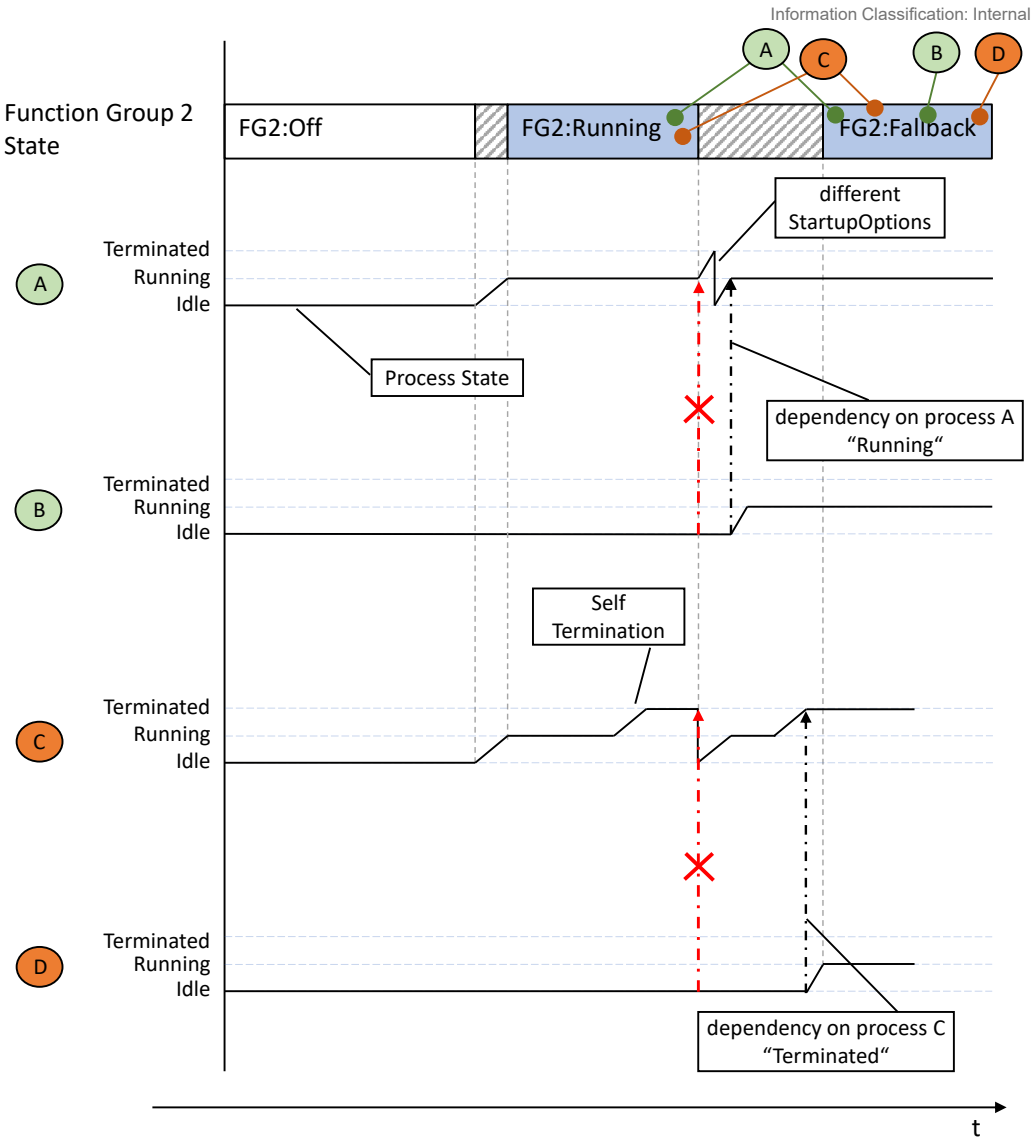

Figure 7.14: Dependency resolution during state change

[SWS_EM_02245] Dependency resolution during state change

Upstream requirements: RS_EM_00101

⌈`Execution Management` shall perform `Execution Dependency` resolution against the `Processes` that are configured for `RequestedState`.⌋

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`.

When considering Figure 7.14 we can imagine a following situation. Should self-terminating Process C indicate it is running, reported kRunning to Execution Management, but fails to terminate due to internal misbehaviour (for instance). Execution Management remains unable to detect any fault in Process C (since it previously reported kRunning), consequently it would be unable to start Process D resulting in a deadlock that blocks Function Group State transition from completion. Projects that use Terminated Execution Dependency must handle this situation outside of Execution Management, for example this can be done in State Management .

Description of Function Group State transition in this chapter may give impression that, it is required to first stop all Processes that are not needed in RequestedState, before you can start any of the Processes that are needed. Please note that this is not the case. Step by step approach of this chapter was chosen to introduce as much clarity as possible, when describing Function Group State transition. Implementers are free to parallelize as much steps (needed for state transition) as possible for a particular implementation.

Execution Management considers a state transition has been performed successfully when the following have occurred:

- Dependency resolution ([SWS_EM_02245]) has been performed.
- All Processes expected to terminate have terminated ([SWS_EM_01060])
- All started ([SWS_EM_01066]) or restarted [SWS_EM_02251]) Reporting Processes have reported kRunning.

Please be aware that [SWS_EM_02315] can also negatively impact the Function Group State transition result.

[SWS_EM_01067] Actions on Completion State Transition

Upstream requirements: RS_EM_00101

⌈On successful completion of a state transition, Execution Management shall set the CurrentState to the RequestedState and report success back to State Management.⌋

[SWS_EM_02315] Unexpected Termination of processes configured for the Requested State during a Function Group State transition

Upstream requirements: RS_EM_00101

⌈In case of Unexpected Termination of a process configured for the RequestedState, Execution Management shall perform the following actions:

1. Stop the Function Group State transition, so State Management can decide how to proceed.
2. log event, if logging is activated

3. Set the CurrentState to Undefined Function Group State.
4. Report kUnexpectedTermination in the ara::exec::StateClient::SetState interface to indicate that the State change request cannot be fulfilled.
5. Return the configured executionError via the ara::exec::StateClient::GetExecutionError interface.

⌋

Please note that [SWS_EM_02315] also applies to Unexpected Self-termination.

[SWS_EM_02316] Unexpected Termination of a process not configured for the Requested State during a Function Group State transition

Upstream requirements: RS_EM_00101

⌈In case of Unexpected Termination of a Process not configured for the RequestedState during a Function Group State, Execution Management shall continue the Function Group State transition and log the event, if logging is activated.⌋

If Process B depends on the termination of Process A during a Function Group State transition (this means that both are configured for the RequestedState), the transition fails if Process A dies unexpectedly before reaching the Terminated process state ([SWS_EM_02315]).

However if a process is only configured for the CurrentState and it dies unexpectedly during the Function Group State transition, then this doesn't stop the Function Group State transition. The Unexpected Termination is logged and normal state transition activities are resumed ([SWS_EM_02316]). The reason to continue the transition is twofold:

- The Process was already marked to be terminated by Execution Management (it was not configured for the RequestedState). It is irrelevant from the State Management point of view if it terminated as expected or due to an internal error.
- Transition to the Off state is always guaranteed to be successful and can be used as a reaction to an error condition.

[SWS_EM_02297] StateClient usage restriction

Upstream requirements: RS_EM_00101

⌈StateClient API shall treat it as a Violation when invoked by a Process with Process.functionClusterAffiliation configured to anything else than STATE_MANAGEMENT.⌋

[SWS_EM_02584] SetState access control

Upstream requirements: RS_EM_00101

⌈`ara::exec::StateClient::SetState` shall return `kInvalidMetaModelIdentifier` if the calling `Process` does not have a `FunctionGroupPortMapping` to the `Function Group` associated with the given `Function Group State`.⌋

If not protected `StateClient` can be used to destabilise `Machine`, see Section 8.2.4 for more details.

Please note that multiple `Processes` with `State Management` responsibilities can be deployed on a single `Machine`. However, to maintain a consistent state of a `Function Group`, it is crucial that a `Function Group` is managed by a single `State Management Process`. If multiple `Processes` were managing a `Function Group`, it would create ambiguity with regards to which `State Management Process` is finally responsible. This could potentially lead to an inconsistent state of a `Function Group`.

[SWS_EM_CONSTR_02560] Function Group shall be controlled by a single State Management process

Upstream requirements: RS_EM_00101

⌈A `Function Group` shall be referenced at most by one `FunctionGroupPortMapping`.⌋

Creating multiple instances of the `StateClient` class inside a `Process` is unnecessary and leads to ambiguity in error handling responsibilities. This duplication makes it unclear which `StateClient` is responsible for the error recovery and should be notified if a `Function Group` transitions to the `Undefined Function Group State` (see [SWS_EM_01309]).

[SWS_EM_02585] Single StateClient instance

Upstream requirements: RS_EM_00101

⌈`ara::exec::StateClient` shall treat the creation of an additional instance as a violation.⌋

7.6 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_SAF_-10008][12]. 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.

7.6.1 Resource Configuration

This section provides an overview on resource assignment to Modelled 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 Modelled 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 `Modelled 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 7.5. If not enough resources are available at any given time to fulfill the maximum resource needs of all running `Modelled Processes`, assuming they are actually used by the `Modelled Processes`, several steps have to be considered:

  - Assignment of resource criticality to `Modelled 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 7.6.3.1.
  - 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 Section 7.6.3.4.

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 `Modelled`

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

[SWS_EM_02102] Memory control

Upstream requirements: RS_EM_00005

⌈Execution Management shall use ResourceGroup.memUsage to configure the maximum amount of RAM available for all Processes in the ResourceGroup before loading any Process from the ResourceGroup.⌋

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

[SWS_EM_02103] CPU usage control

Upstream requirements: RS_EM_00005

⌈Execution Management shall use ResourceGroup.cpuUsage to configure the maximum amount of CPU time available for all Processes in each ResourceGroup before loading any Process from the ResourceGroup.⌋

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...).

Because scheduling is done in very different ways depending on the Operating System, the specific algorithm for scheduling as well as limiting the CPU usage is not described [SWS_OSI_02002].

The intention of ResourceGroup is that limits are never reached and the ResourceGroup limits shall be configured by the integrator, based on measurement, not worst-case execution time.

7.6.2 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 Section 7.6.3.1. For RAM monitoring see Section 7.6.3.4. 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 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 Section 7.6.1), one of the following alternatives is valid:

- The resource limit violation is considered as a failure and recovery actions may need to be initiated. Therefore the specific violation gets reported to the State Management, which then starts recovery actions.
- 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.

7.6.3 Application-level Resource Configuration

We need to be able to configure minimum, guaranteed resources (RAM, computing time) and maximum resources.

7.6.3.1 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.

7.6.3.2 Core Affinity

[SWS_EM_02104] Core affinity

Upstream requirements: RS_EM_00008

⌈Execution Management shall configure the Core affinity of the Process initial thread (restricting it to a sub-set of cores in the system) based on the configuration ProcessToMachineMapping.shallRunOn and ProcessToMachineMapping.shallNotRunOn.⌋

Requirement [SWS_EM_02104] together with [constr_1677] (which ensures mutual exclusion of ProcessToMachineMapping.shallRunOn and ProcessToMachineMapping.shallNotRunOn for a specific process), 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.

7.6.3.3 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] Scheduling policy

Upstream requirements: RS_EM_00002

⌈Execution Management shall configure the process scheduling policy (when launching a Process) based on the relevant configuration StartupConfig.schedulingPolicy.⌋

For the detailed definitions of these policies, refer to [13]. 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] Scheduling FIFO**

  Upstream requirements: RS_EM_00002

  ⌈Execution Management shall be able to configure FIFO scheduling using policy `SCHED_FIFO`.⌋

- **[SWS_EM_01042] Scheduling Round-Robin**

  Upstream requirements: RS_EM_00002

  ⌈Execution Management shall be able to configure round-robin scheduling using policy SCHED_RR.⌋

- **[SWS_EM_01043] Scheduling Other**

  Upstream requirements: RS_EM_00002

  ⌈Execution Management shall be able to configure non real-time scheduling using policy SCHED_OTHER.⌋

Note that the Scheduling Policies specified here are the minimal set. Depending on the OS there may be more Scheduling Policies configurable.

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] Scheduling priority

Upstream requirements: RS_EM_00002

⌈Execution Management shall support the configuration of a scheduling priority when launching a Process based on the relevant configuration StartupConfig.schedulingPriority.⌋

The available priority range and actual meaning of the scheduling priority depends on the selected scheduling policy, see [constr_1692] [2], [TPS_MANI_01061] and [TPS_MANI_01188] [2].

7.6.3.3.1 Resource Management

In general, for deterministic behavior the required computing time is guaranteed and violations are treated as errors, 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 an optional 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.

[SWS_EM_02106] ResourceGroup assignment

Upstream requirements: RS_EM_00005

⌈Execution Management shall configure the Process according to its ResourceGroup membership.⌋

7.6.3.4 Memory Budget and Monitoring

Processes require memory for their execution (e.g. 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 also impacts the process performance: 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.

In order to ensure that sufficient memory is available at the start and for the whole duration of the of a Process, some properties may need to be defined for each Process.

[SWS_EM_02108] Maximum memory usage

Upstream requirements: RS_EM_00005

⌈Execution Management shall configure the Maximum memory usage of the Process according to the configuration item Process.stateDependentStartupConfig.resourceConsumption.memoryUsage.⌋

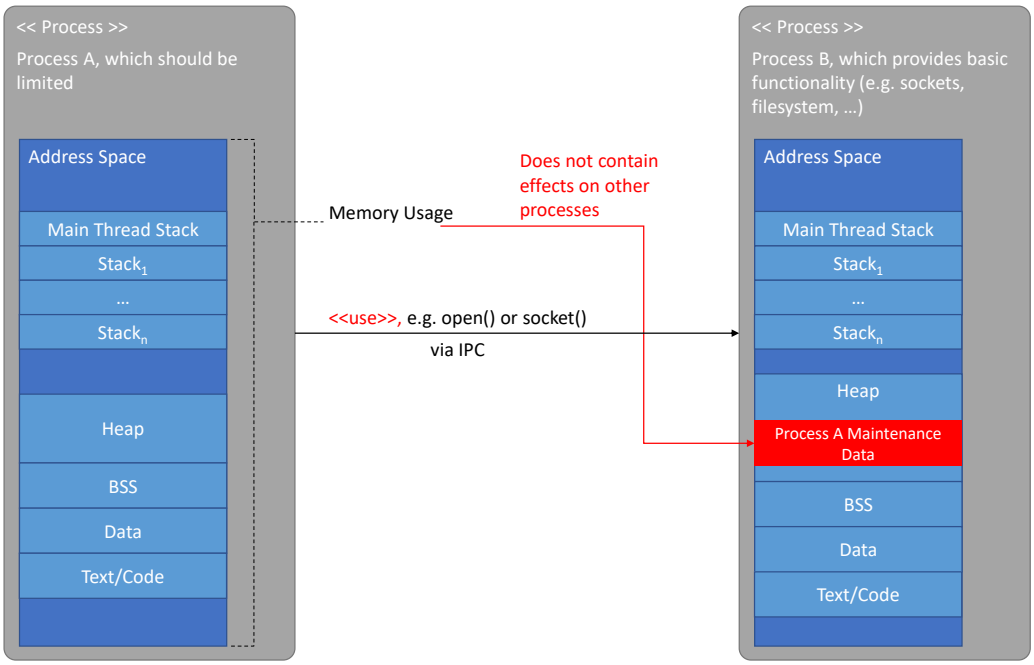

Figure 7.15: Memory Usage.

The `resourceConsumption.memoryUsage` is the amount of memory which can be allocated by the `Process` itself. Please be aware, depending on the OS and its configuration this does not necessarily contain all the memory the process can allocate within the system. For example in an OS where common functionality like a file system is implemented on process level, the restricted `Process` might still lead to memory allocations within the `Process` providing the file system.

On POSIX OS the memory limit is typically restricted by the resource RLIMIT_AS.

[SWS_EM_02109] process pre-mapping

Upstream requirements: RS_EM_00005

⌈`Execution Management` shall pre-map a `Process` if `Process.preMapping` is set to true.⌋

Fully pre-mapping a `Modelled Process` ensures that code and data execution is not going to be delayed at its first execution by demand-loading. This approach not only supports predictable timing 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 `Modelled Process` may not be loadable anymore.

7.6.3.5 Working Folder

The working folder of a process is not defined by configuration but rather is deliberately left as an implementation-specific element. The required PSE51 POSIX profile does not define that an `Adaptive Application` may use the path or file argument for any function using a file pathname (e.g., `open`), instead only to specify the name of the object without any file system semantics implied.

The PSE51 POSIX profile does not require the existence of a file system. Consequently, paths in `Adaptive Applications` merely identify objects (e.g. in calls to `open()` or `stat()`). The usage of sub-parts of a given path (e.g. "`/data`" when "`/data/config.dat`" was given) is implementation-defined.

7.7 Fault Tolerance

7.7.1 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 [14].

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.

7.7.2 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 failure.

It is therefore necessary not to specify only 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 overall 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.

7.7.3 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.

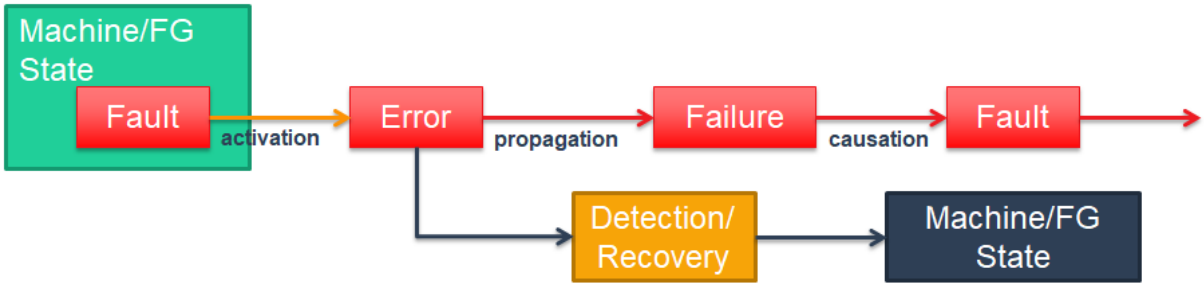

Figure 7.16: General Fault Tolerance scheme.

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.

7.7.4 Execution Management internal Error handling

From System design point of view it is useful to have an Execution Management/OS internal Unrecoverable State, which can be entered by Execution Management when it has no other course of action. The Unrecoverable State is only triggered by Execution Management.

[SWS_EM_02032] Behavior on entry to the Unrecoverable State

Upstream requirements: RS_EM_00150

⌈On entry to the Unrecoverable State,Execution Management shall invoke a pre-cleanup action.⌋

[SWS_EM_02033] Behavior after execution of the pre-cleanup action

Upstream requirements: RS_EM_00150

⌈After execution of the pre-cleanup action, all Processes managed by Execution Management shall be shutdown.⌋

[SWS_EM_02034] Behavior after termination of all processes managed by Execution Management

Upstream requirements: RS_EM_00150

⌈After all processes managed by Execution Management are terminated, a post-cleanup action shall be called.⌋

The mechanism for invoking pre- and post-cleanup function is Platform specific. There is no requirement on which actions should be taken at each stage.

It is not possible to give an exhaustive of list of when the Unrecoverable State is entered. Potential examples when the Unrecoverable State should be entered include:

- The underlying OS is not functioning as expected – for example failure of SIGKILL (i.e. Execution Management cannot kill processes).
- Execution Management has lost the ability to read the processed manifest, i.e. nothing can be started / stopped.
- Execution Management cannot deliver responses (i.e. result of the requested Function Group state transitions) to State Management. Essentially Execution Management will never respond back to SM for technical reasons.
- Trusted platform configuration cannot be read meaning Execution Management does not know it should run in a strictMode or monitorMode.

Note: Unrecoverable State should not be entered if Execution Management can normally communicate with State Management – in this case it is State Management's responsibility to handle system errors (i.e. failed startup attempts).

7.8 Security

7.8.1 Trusted Platform

From a security perspective, it is essential that all software executed on the Adaptive Platform is trusted, i.e. the integrity and authenticity of the software is ensured. Execution Management - as the entity responsible for Process creation - has to take over this task.

A key requirement for a trusted Adaptive Platform is a Trust Anchor on the Machine that is authentic by definition (hence that alternative name, "root of trust"). A Trust Anchor is often realized as a public key stored in a secure environment, e.g. in non-modifiable persistent memory or in an HSM. The trust has to be passed to Execution Management by appropriate means, e.g. by a chain of trust. If the Machine does not exhibit a Trust Anchor, it cannot be ensured that the Adaptive Platform is trusted.

[SWS_EM_02299] Availability of a Trust Anchor

Upstream requirements: RS_EM_00014

⌈If there is no Trust Anchor available on the Machine, the following requirements may be ignored: [SWS_EM_02300], [SWS_EM_02301], [SWS_EM_02302], [SWS_EM_02303], [SWS_EM_02305], [SWS_EM_02306], [SWS_EM_02307], [SWS_EM_02308], [SWS_EM_02309].⌋

There are many ways to verify the integrity and authenticity of the Adaptive Platform. A Trusted Platform can be realized e.g. (but not limited to) by

- Verification of the complete Ramdisk by the Bootloader
- Verification of individual Executables and data files, e.g. using OS-functionalities or a trusted third-party process
- Verification of individual memory pages upon being loaded, e.g. using OS-functionalities or a trusted third-party process

[SWS_EM_02300] Integrity and Authenticity of Machine configuration

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈Execution Management shall ensure that the integrity and authenticity of Machine information from the processed Manifests are checked before use.⌋

[SWS_EM_02301] Integrity and Authenticity of each Executable

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈Execution Management shall ensure that for every Process that is about to be started, the integrity and authenticity of the Executable itself are checked.⌋

[SWS_EM_02302] Integrity and Authenticity of shared objects

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈`Execution Management` shall ensure that for every `Process` that is about to be started, the integrity and authenticity of each related shared object are checked.⌋

[SWS_EM_02303] Integrity and Authenticity of processed Execution Manifest configurations

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈`Execution Management` shall ensure that for every `Process` that is about to be started, the integrity and authenticity of its corresponding processed `Manifests` are checked.⌋

The information validated by [SWS_EM_02303] includes all manifest information, e.g. Service Instance information, and not just the information directly used by `Execution Management`.

From a security perspective, the rationale for choosing these items is as follows:

- `Executables`: Modifying the Executable itself allows an attacker to execute arbitrary code on the machine;
- Manifests: `Machine Manifests`, `Execution Manifests` and `Service Instance Manifests` describe what and how something should be executed and are thus an obvious attack vector on the Adaptive Platform;
- Shared Objects: Shared objects can either contain code that is executed within the context of the `Process` or data that (potentially) influences the execution of a `Process` accessing this data. A modified shared object could consequently be used to compromise the system.

In order to establish a `Trusted Platform`, it must be ensured that only trusted software is launched. Therefore, a system designer has to ensure that `Execution Management` is started authentically. For instance, this could be realized by a chain of trust as described in [15].

`Execution Management` in turn has to ensure that all `Executable` code on the Adaptive Platform is authenticated before being executed. The complete authenticated start-up sequence looks like this:

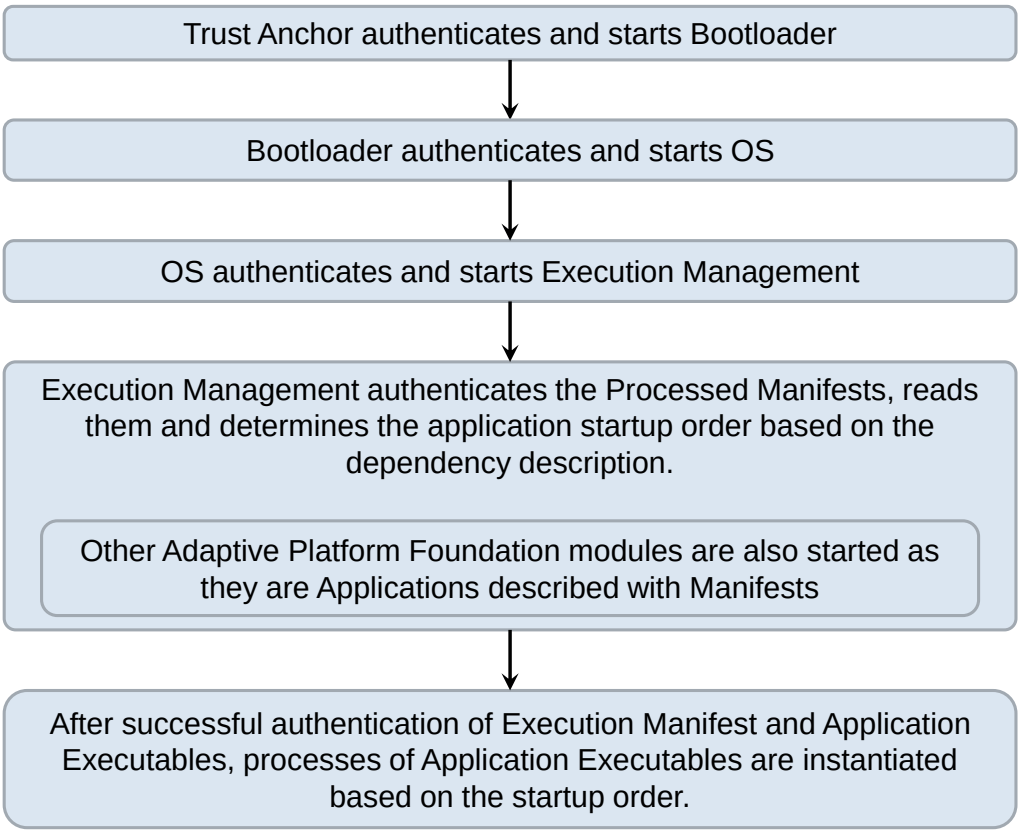

Figure 7.17: Authenticated start-up sequence

The integrity and authenticity of persistent data stored by applications is not considered here. The `Functional Cluster` Persistency takes care of the integrity of this data.

7.8.1.1 Handling of failed authenticity checks

If the integrity and authenticity has been verified successfully, the system has to continue with its regular start-up process. If the integrity and authenticity check has failed, however, `Execution Management` offers a configuration option on how to proceed with the start-up process.

[SWS_EM_02305] Failed authenticity checks

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈`Execution Management` shall select the trusted platform mode based on the value of `Machine.trustedPlatformExecutableLaunchBehavior`.⌋

The configuration of the three modes is done via the `trustedPlatformExecutableLaunchBehavior` attribute within the `Processed Manifest`. The configuration option is only allowed to be processed after the integrity and authenticity of the relevant `Processed Manifest` has been verified.

These three modes of the Machine.trustedPlatformExecutableLaunchBehavior are:

- monitorMode
- noTrustedPlatformSupport
- strictMode

[SWS_EM_02306] Launch Behavior Validation

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈Execution Management shall stop the start-up sequence of the Adaptive Platform if the integrity or authenticity check of the Processed Manifest containing the trustedPlatformExecutableLaunchBehavior selection has failed and switch to the unrecoverable state.⌋

The integrity and authenticity check applies to all trusted platform modes; to do otherwise would leave the system open to attacks that maliciously corrupt the Manifest information. Reaction to a failure is limited as, by definition, no Adaptive Application other than Execution Management are running and hence are restricted to implementation defined actions such as OS-level logging.

7.8.1.1.1 Monitor Mode

In monitorMode, the integrity and authenticity checks are performed, but the start-up process is not affected. Hence, the Adaptive Platform starts up even if the file system has been compromised.

[SWS_EM_02556] Monitor Mode

Upstream requirements: RS_EM_00014, RS_EM_00015

⌈In monitorMode, if a check ([SWS_EM_02300], [SWS_EM_02301], [SWS_EM_02302] and [SWS_EM_02303]) fails then Execution Management shall log the failure and continue regular startup of the Adaptive Platform.⌋

monitorMode is useful when the integrator wants the system to keep running, even if the platform is not considered trusted. In this case, the integrator might use additional measures outside the scope of Adaptive AUTOSAR, like e.g. restricted key access when using an HSM that supports this feature.

monitorMode is also useful during development phase, when frequent changes on the Adaptive Platform are performed and keeping the authentication tag (e.g. signatures) valid is a tedious task.

7.8.1.1.2 Strict Mode

In strictMode, the Adaptive Platform ensures that no Processes are executed, where the integrity and authenticity of the corresponding Executable, manifests or linked library could not be verified.

[SWS_EM_02307] Strict Mode - Execution manifest

Upstream requirements: RS_EM_00014

⌈In strictMode, Execution Management shall not initiate the execution of an Executable if the integrity or authenticity check of the corresponding processed Execution Manifest has failed.⌋

[SWS_EM_02308] Strict Mode - Service Instance manifests

Upstream requirements: RS_EM_00014

⌈In strictMode, Execution Management shall not initiate the execution of an Executable if the integrity or authenticity check of at least one of the corresponding processed Service Instance Manifests has failed.⌋

[SWS_EM_02309] Strict Mode - Executables

Upstream requirements: RS_EM_00014

⌈In strictMode, Execution Management shall start a Process only if the integrity and authenticity of the corresponding Executable was successfully verified.⌋

Executable code can be provided by executables and by statically linked shared objects linked by the executable. Execution Management cannot determine dynamically linked shared objects and thus these needs to be validated through an alternative, implementation specific, mechanism.

Example: Consider an Adaptive Platform in strictMode. Execution Management has started several Executables after successfully verifying the integrity and authenticity of the Executable, its related shared objects and its processed Execution Manifest. Now, Execution Management wants to start another Executable, where the authenticity check has failed. Execution Management does not launch this Executable, because it is not trusted. The other Executables that passed the authenticity check may however continue to run. When Execution Management attempts to start another Executable it can be started as long as all authenticity checks are passed.

7.8.2 Identity and Access Management

Following the "Principle of Least Privilege", Identity and Access Management (IAM) [16] was introduced in the Adaptive Platform. IAM allows to assign permissions to Modelled Processes for accessing public Interfaces from Functional Clusters. Hence, Modelled Processes have to be identifiable during runtime in order to lookup and enforce permissions accordingly.

Execution Management starts Processes based on Modelled Processes. Hence Execution Management is able to maintain the association between the two. Execution Management supports IAM by revealing information about this association. This allows IAM to authenticate processes during runtime with the help of the operating system and Execution Management.

[SWS_EM_02400] Properties of IAM-configuration assigned to processes

Upstream requirements: RS_EM_00111, RS_EM_00015

⌈Execution Management shall associate the *identity* of a specific Modelled Process with the identity of the corresponding Process during Process creation.⌋

The form of identity is implementation specifc but could, for example, be the process identifier, a cryptographic token, user ID, etc.

Based on implementation requirements, Execution Management may expose interfaces that allow IAM to retrieve information about the association between Process and Modelled Process identity. The exact form of this interface is implementation defined.

8 API specification

This chapter provides a reference of the APIs defined by this functional cluster. The API is described in the following chapters in tables. Table 8.1 explains the content that is described in such an API table.

Kind:	Defines the kind of the declaration that this API table describes. The following values are supported: • class (Declaration of a class) • function (Declaration of a member or non-member function) • struct (Declaration of a structure) • type alias (Declaration of a type alias) • enumeration (Declaration of an enumeration) • variable (Declaration of a variable)	
Header File:	Defines the header file to be included according to [SWS_CORE_90001]	
Forwarding Header File:	Defines the forwarding header file to be included according to [SWS_CORE_90001]	
Scope:	Defines the scope that may be a namespace (in case of a class or non-member function) or a class declaration (in case of a member)	
Symbol:	Entity name	
Thread Safety:	Defines whether a function is thread-safe, not thread-safe, or conditional according to [SWS_CORE_13200] and [SWS_CORE_13202]	
Syntax:	Description of C++ syntax	
Template Param:	Template parameter (0..*)	Template parameter(s) used to parametrize the template
Parameters (in):	Parameter declaration (0..*)	Parameter(s) that are passed to the function
Parameters (out):	Parameter declaration (0..*)	Parameter(s) that are returned to the caller
Return Value:	Return type	Type of the value that the function returns
Exception Safety:	Defines whether a function is exception-safe, not exception safe or conditionally exception safe	
Exceptions:	List of exceptions that may be thrown from the function	
Violations:	List of violations that may occur in the function	
Errors:	Error type (0..*)	List of defined error codes that may be returned by the function with their recoverability class defined in [RS_AP_00160]. APIs can be extended with vendor-specific error codes. These are not part of the AUTOSAR SWS specifications
Description:	Brief description of the function	

Table 8.1: Explanation of an API table

8.1 Type Definitions

8.1.1 ExecutionState

[SWS_EM_02000] Definition of API enum ara::exec::ExecutionState

Upstream requirements: RS_EM_00103, RS_AP_00134, RS_AP_00125, RS_AP_00143, RS_AP_00129

⌈

Kind:	enumeration	
Header file:	#include "ara/exec/execution_client.h"	
Forwarding header file:	#include "ara/exec/exec_fwd.h"	
Scope:	`namespace ara::exec`	
Symbol:	ExecutionState	
Underlying type:	std::uint32_t	
Syntax:	`enum class ExecutionState : std::uint32_t {...};`	
Values:	kRunning= 0	After a Process has been started by Execution Management, it reports ExecutionState kRunning.
Description:	Defines the internal states of a Process. Scoped Enumeration of uint8_t .	

⌋

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 7.2 and Figure 7.12, 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 ([SWS_EM_01004]). The Terminating state starts when termination is requested by `Execution Management` and ends when the `Process` terminates ([SWS_EM_01404]). For the reasons mentioned, `Execution Management` assumes that `Process` is in initializing state until `kRunning` will be reported by it.

8.1.2 ExecutionError

[SWS_EM_02541] Definition of API type ara::exec::ExecutionError

Upstream requirements: RS_EM_00101, RS_AP_00122, RS_AP_00154

⌈

Kind:	type alias
Header file:	#include "ara/exec/execution_error_event.h"
Scope:	`namespace ara::exec`
Symbol:	ExecutionError

▽

△

Syntax:	`using ExecutionError = std::uint32_t;`
Description:	Represents the execution error.

⌋

8.1.3 ExecutionErrorEvent

[SWS_EM_02544] Definition of API class ara::exec::ExecutionErrorEvent

Upstream requirements: RS_EM_00101, RS_AP_00116, RS_AP_00122, RS_AP_00124, RS_AP_00140, RS_AP_00154

⌈

Kind:	struct
Header file:	#include "ara/exec/execution_error_event.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	ExecutionErrorEvent
Syntax:	`struct ExecutionErrorEvent final {...};`
Description:	Represents an execution error event which happens in a Function Group.

⌋

8.1.3.1 ExecutionErrorEvent::executionError

[SWS_EM_02545] Definition of API variable ara::exec::ExecutionErrorEvent::executionError

Upstream requirements: RS_EM_00101, RS_AP_00124

⌈

Kind:	variable
Header file:	#include "ara/exec/execution_error_event.h"
Scope:	`struct ara::exec::ExecutionErrorEvent`
Symbol:	executionError
Type:	`ExecutionError`
Syntax:	`ExecutionError executionError;`
Description:	The execution error of the Process which unexpectedly terminated .

⌋

8.1.3.2 ExecutionErrorEvent::functionGroup

[SWS_EM_02546] Definition of API variable ara::exec::ExecutionErrorEvent::functionGroup

Upstream requirements: RS_EM_00101, RS_AP_00124

⌈

Kind:	variable
Header file:	#include "ara/exec/execution_error_event.h"
Scope:	`struct ara::exec::ExecutionErrorEvent`
Symbol:	functionGroup
Type:	`FunctionGroup`
Syntax:	`FunctionGroup functionGroup;`
Description:	The function group in which the error occurred .

⌋

8.2 Class Definitions

As specified in [9] AUTOSAR Adaptive Platform requires initialization and deinitialization, see [SWS_CORE_10001] and [SWS_CORE_10002]. Usage of `Execution Management` API before a call to `ara::core::Initialize`, or after `ara::core::Deinitialize` will result in implementation defined behavior, see [SWS_CORE_90022] and [SWS_CORE_15005].

[SWS_EM_02557] Initialization and deinitialization of Execution Management API

Upstream requirements: RS_AP_00155, RS_AP_00149

⌈`Execution Management` shall implement all actions necessary for the initialization and deinitialization of `ara::exec` APIs within the standard `ara::core::Initialize` and `ara::core::Deinitialize` APIs.⌋

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] Definition of API class ara::exec::ExecutionClient

Upstream requirements: RS_EM_00103, RS_AP_00154, RS_AP_00150

⌈

Kind:	class
Header file:	#include "ara/exec/execution_client.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	ExecutionClient
Syntax:	`class ExecutionClient final {...};`
Description:	Class to implement operations on Execution Client.
Notes:	To eventually implement the Named Constructor Idiom, the developer may either make the default constructor private or delete it and define a non-default constructor.

⌋

8.2.1.1 ExecutionClient::ExecutionClient

[SWS_EM_02560] Definition of API function ara::exec::ExecutionClient::ExecutionClient

Upstream requirements: RS_EM_00103, RS_AP_00114, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00151

⌈

Kind:	function	
Header file:	#include "ara/exec/execution_client.h"	
Scope:	`class ara::exec::ExecutionClient`	
Syntax:	`ExecutionClient (std::function< void()> terminationHandler) noexcept(false);`	
Parameters (in):	terminationHandler	Callback which is called if ExecutionClient receives SIGTERM signal. The callback is executed in a background thread. A typical implementation of this callback will set a global flag (and potentially unblock other threads) to perform a graceful termination. Lifetime: it is expected that terminationHandler remains callable, during entire lifetime of the ExecutionClient instance. This is especially important if, terminationHandler is bound to an instance of a class (e.g. using std::bind).
Exception Safety:	not exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kNoCommunication	--
		Communication error occurred.
	ara::exec::ExecErrc::kInvalidArgument	--
		Given terminationHandler doesn't contain a callable function.
Description:	Regular constructor for ExecutionClient.	

⌋

8.2.1.2 ExecutionClient::Create

[SWS_EM_02562] Definition of API function ara::exec::ExecutionClient::Create

Upstream requirements: RS_AP_00119, RS_AP_00120, RS_AP_00128, RS_AP_00139, RS_AP_00144, RS_EM_00103

⌈

Kind:	function	
Header file:	#include "ara/exec/execution_client.h"	
Scope:	`class ara::exec::ExecutionClient`	
Syntax:	`static ara::core::Result< ExecutionClient > Create (std::function< void()> terminationHandler) noexcept;`	
Parameters (in):	terminationHandler	Callback which is called if ExecutionClient receives SIGTERM signal. The callback is executed in a background thread. A typical implementation of this callback will set a global flag (and potentially unblock other threads) to perform a graceful termination. Lifetime: it is expected that terminationHandler remains callable, during entire lifetime of the ExecutionClient instance. This is especially important if, terminationHandler is bound to an instance of a class (e.g. using std::bind).
Return value:	ara::core::Result< ExecutionClient >	a result that contains either a ExecutionClient object or an error.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kNoCommunication	--
		Communication error occurred.
	ara::exec::ExecErrc::kInvalidArgument	--
		Given terminationHandler doesn't contain a callable function.
Description:	Named constructor for ExecutionClient.	
Notes:	This named constructor may call a constructor defined by the developer.	

⌋

8.2.1.3 ExecutionClient::~ExecutionClient

[SWS_EM_02002] Definition of API function ara::exec::ExecutionClient::~ExecutionClient

Upstream requirements: RS_AP_00134, RS_AP_00145, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/execution_client.h"
Scope:	`class ara::exec::ExecutionClient`
Syntax:	`~ExecutionClient () noexcept;`
Exception Safety:	exception safe

▽

△

Thread Safety:	implementation defined
Description:	noexcept destructor
Notes:	Since ExecutionClient overtake the responsibility for handling SIGTERM signal, it should reset SIGTERM handler back to its original value, when destructor is called.

⌋

8.2.1.4 ExecutionClient::ExecutionClient (deleted Copy Constructor)

[SWS_EM_02563] Definition of API function ara::exec::ExecutionClient::ExecutionClient

Upstream requirements: RS_AP_00145, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/execution_client.h"
Scope:	`class ara::exec::ExecutionClient`
Syntax:	`ExecutionClient (const ExecutionClient &)=delete;`
Description:	Suppress default copy construction for ExecutionClient.

⌋

8.2.1.5 ExecutionClient::operator= (deleted Copy assignment operator)

[SWS_EM_02564] Definition of API function ara::exec::ExecutionClient::operator=

Upstream requirements: RS_AP_00145, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/execution_client.h"
Scope:	`class ara::exec::ExecutionClient`
Syntax:	`ExecutionClient & operator= (const ExecutionClient &)=delete;`
Description:	Suppress default copy assignment for ExecutionClient.

⌋

8.2.1.6 ExecutionClient::ExecutionClient (use of default move constructor)

[SWS_EM_02580] Definition of API function ara::exec::ExecutionClient::ExecutionClient

Upstream requirements: RS_AP_00145, RS_AP_00151, RS_EM_00103

⌈

Kind:	function	
Header file:	#include "ara/exec/execution_client.h"	
Scope:	`class ara::exec::ExecutionClient`	
Syntax:	`ExecutionClient (ExecutionClient &&rval) noexcept;`	
Parameters (in):	rval	reference to move
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Intentional use of default move constructor for ExecutionClient.	

⌋

8.2.1.7 ExecutionClient::operator= (use of default move assignment)

[SWS_EM_02581] Definition of API function ara::exec::ExecutionClient::operator=

Upstream requirements: RS_AP_00145, RS_AP_00151, RS_EM_00103

⌈

Kind:	function	
Header file:	#include "ara/exec/execution_client.h"	
Scope:	`class ara::exec::ExecutionClient`	
Syntax:	`ExecutionClient & operator= (ExecutionClient &&rval) noexcept;`	
Parameters (in):	rval	reference to move
Return value:	ExecutionClient &	the new reference
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Intentional use of default move assignment for ExecutionClient.	

⌋

8.2.1.8 ExecutionClient::ReportExecutionState

[SWS_EM_02003] Definition of API function ara::exec::ExecutionClient::Report ExecutionState

Upstream requirements: RS_EM_00103, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00128, RS_AP_00139

⌈

Kind:	function	
Header file:	#include "ara/exec/execution_client.h"	
Scope:	`class ara::exec::ExecutionClient`	
Syntax:	`ara::core::Result< void > ReportExecutionState (ExecutionState state) noexcept;`	
Parameters (in):	state	Value representing the current Process state, that should be reported.
Return value:	ara::core::Result< void >	An instance of ara::core::Result. The instance holds an ErrorCode containing either one of the specified errors or a void-value.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kNo Communication	--
		Communication error between Application and Execution Management, e.g. unable to get confirmation that report was received.
	ara::exec::ExecErrc::k InvalidTransition	--
		Invalid transition request (e.g. to Running when already in Running state)
Description:	Interface for a Process to report its internal state to Execution Management.	

⌋

8.2.2 FunctionGroup class

An instance of this class will represent `Function Group` defined inside meta-model (ARXML). This class is intended to be an implementation specific representation, of information inside meta-model. Once created based on ARXML path, its internal value stays bounded to it for entire lifetime of a object.

[SWS_EM_02263] Definition of API class ara::exec::FunctionGroup

Upstream requirements: RS_EM_00101, RS_AP_00154, RS_AP_00150

⌈

Kind:	class
Header file:	#include "ara/exec/function_group.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"

▽

△

Scope:	`namespace ara::exec`
Symbol:	FunctionGroup
Syntax:	`class FunctionGroup final {...};`
Description:	Class representing Function Group defined in meta-model (ARXML).
Notes:	Once created based on ARXML path, it's internal value stay bounded to it for entire lifetime of an object.

⌋

8.2.2.1 FunctionGroup::FunctionGroup

[SWS_EM_02586] Definition of API function ara::exec::FunctionGroup::FunctionGroup

Upstream requirements: RS_AP_00137, RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group.h"	
Scope:	`class ara::exec::FunctionGroup`	
Syntax:	`FunctionGroup (const ara::core::InstanceSpecifier &instance) noexcept;`	
Parameters (in):	instance	instance specifier to the RPortPrototype of a StateClientInterface.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Creates an instance of FunctionGroup.	

⌋

8.2.2.2 FunctionGroup::FunctionGroup (Default Constructor)

[SWS_EM_02321] Definition of API function ara::exec::FunctionGroup::FunctionGroup

Upstream requirements: RS_EM_00101

⌈

Kind:	function
Header file:	#include "ara/exec/function_group.h"
Scope:	`class ara::exec::FunctionGroup`
Syntax:	`FunctionGroup ()=delete;`

▽

△

Description:	Default constructor.
Notes:	Default constructor is deleted in favour of regular constructor.

⌋

8.2.2.3 FunctionGroup::FunctionGroup (Copy Constructor)

[SWS_EM_02322] Definition of API function ara::exec::FunctionGroup::FunctionGroup

Upstream requirements: RS_AP_00147, RS_EM_00101

⌈

Kind:	function
Header file:	#include "ara/exec/function_group.h"
Scope:	`class ara::exec::FunctionGroup`
Syntax:	`FunctionGroup (const FunctionGroup &other)=delete;`
Description:	Copy constructor.
Notes:	To prevent problems with resource allocations during copy operation, this class is non-copyable.

⌋

8.2.2.4 FunctionGroup::FunctionGroup (Move Constructor)

[SWS_EM_02328] Definition of API function ara::exec::FunctionGroup::FunctionGroup

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group.h"	
Scope:	`class ara::exec::FunctionGroup`	
Syntax:	`FunctionGroup (FunctionGroup &&other) noexcept;`	
Parameters (in):	other	FunctionGroup instance to move to a newly constructed object.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Move constructor.	

⌋

8.2.2.5 FunctionGroup::operator= (Copy assignment operator)

[SWS_EM_02327] Definition of API function ara::exec::FunctionGroup::operator=

Upstream requirements: RS_AP_00147, RS_EM_00101

⌈

Kind:	function
Header file:	#include "ara/exec/function_group.h"
Scope:	`class ara::exec::FunctionGroup`
Syntax:	`FunctionGroup & operator= (const FunctionGroup &other)=delete;`
Description:	Copy assignment operator.
Notes:	To prevent problems with resource allocations during copy operation, this class is non-copyable.

⌋

8.2.2.6 FunctionGroup::operator= (Move assignment operator)

[SWS_EM_02329] Definition of API function ara::exec::FunctionGroup::operator=

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group.h"	
Scope:	`class ara::exec::FunctionGroup`	
Syntax:	`FunctionGroup & operator= (FunctionGroup &&other) noexcept;`	
Parameters (in):	other	FunctionGroup instance to move to this object.
Return value:	FunctionGroup &	reference to the object on which the move assignment operator was invoked
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Move assignment operator.	

⌋

8.2.2.7 FunctionGroup::~FunctionGroup

[SWS_EM_02266] Definition of API function ara::exec::FunctionGroup::~FunctionGroup

Upstream requirements: RS_EM_00101

⌈

Kind:	function
Header file:	#include "ara/exec/function_group.h"
Scope:	`class ara::exec::FunctionGroup`
Syntax:	`~FunctionGroup () noexcept;`
Exception Safety:	exception safe
Thread Safety:	implementation defined
Description:	Destructor of the FunctionGroup instance.

⌋

8.2.2.8 FunctionGroup::operator==

[SWS_EM_02267] Definition of API function ara::exec::FunctionGroup::operator==

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group.h"	
Scope:	`class ara::exec::FunctionGroup`	
Syntax:	`bool operator== (const FunctionGroup &other) const noexcept;`	
Parameters (in):	other	FunctionGroup instance to compare this one with.
Return value:	bool	true in case both FunctionGroups are representing exactly the same meta-model element, false otherwise.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	eq operator to compare with other FunctionGroup instance.	

⌋

8.2.2.9 FunctionGroup::operator!=

[SWS_EM_02268] Definition of API function ara::exec::FunctionGroup::operator!=

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group.h"	
Scope:	`class ara::exec::FunctionGroup`	
Syntax:	`bool operator!= (const FunctionGroup &other) const noexcept;`	
Parameters (in):	other	FunctionGroup instance to compare this one with.
Return value:	bool	false in case both FunctionGroups are representing exactly the same meta-model element, true otherwise.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	uneq operator to compare with other FunctionGroup instance.	

⌋

8.2.3 FunctionGroupState class

An instance of this class will represent `Function Group State` defined inside meta-model (ARXML). This class is intended to be an implementation specific representation, of information inside meta-model. Once created based on ARXML path, its internal value stays bounded to it for entire lifetime of a object.

[SWS_EM_02269] Definition of API class ara::exec::FunctionGroupState

Upstream requirements: RS_EM_00101, RS_AP_00154, RS_AP_00150, RS_AP_00129

⌈

Kind:	class
Header file:	#include "ara/exec/function_group_state.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	FunctionGroupState
Syntax:	`class FunctionGroupState final {...};`
Description:	Class representing Function Group State defined in meta-model (ARXML).
Notes:	Once created based on ARXML path, it's internal value stay bounded to it for entire lifetime of an object.

⌋

8.2.3.1 FunctionGroupState::FunctionGroupState

[SWS_EM_02324] Definition of API function ara::exec::FunctionGroupState::FunctionGroupState

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`FunctionGroupState (const FunctionGroup &functionGroup, ara::core::StringView state) noexcept;`	
Parameters (in):	functionGroup	the FunctionGroup instance the state shall be connected with.
	state	short name of the ModeDeclaration which represents the Function Group State.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Creates an instance of FunctionGroupState.	

⌋

8.2.3.2 FunctionGroupState::FunctionGroupState (Copy Constructor)

[SWS_EM_02325] Definition of API function ara::exec::FunctionGroupState::FunctionGroupState

Upstream requirements: RS_EM_00101, RS_AP_00145

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`FunctionGroupState (const FunctionGroupState &other) noexcept;`	
Parameters (in):	other	FunctionGroupState instance to be copied
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Copy constructor.	

⌋

8.2.3.3 FunctionGroupState::FunctionGroupState (Move Constructor)

[SWS_EM_02331] Definition of API function ara::exec::FunctionGroupState::FunctionGroupState

Upstream requirements: RS_EM_00101, RS_AP_00145, RS_AP_00151

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`FunctionGroupState (FunctionGroupState &&other) noexcept;`	
Parameters (in):	other	FunctionGroupState instance to be moved to a newly constructed object.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Move constructor.	

⌋

8.2.3.4 FunctionGroupState::operator= (Copy assignment operator)

[SWS_EM_02330] Definition of API function ara::exec::FunctionGroupState::operator=

Upstream requirements: RS_EM_00101, RS_AP_00153, RS_AP_00145

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`FunctionGroupState & operator= (const FunctionGroupState &other) & noexcept;`	
Parameters (in):	other	FunctionGroupState instance to be copied
Return value:	FunctionGroupState &	reference to the object on which the copy assignment operator was invoked
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Copy assignment operator.	

⌋

8.2.3.5 FunctionGroupState::operator= (Move assignment operator)

[SWS_EM_02332] Definition of API function ara::exec::FunctionGroupState::operator=

Upstream requirements: RS_EM_00101, RS_AP_00153, RS_AP_00145, RS_AP_00151

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`FunctionGroupState & operator= (FunctionGroupState &&other) & noexcept;`	
Parameters (in):	other	FunctionGroupState instance to move to this object.
Return value:	FunctionGroupState &	reference to the object on which the move assignment operator was invoked
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Move assignment operator.	

⌋

8.2.3.6 FunctionGroupState::~FunctionGroupState

[SWS_EM_02272] Definition of API function ara::exec::FunctionGroupState::~FunctionGroupState

Upstream requirements: RS_EM_00101, RS_AP_00134, RS_AP_00145

⌈

Kind:	function
Header file:	#include "ara/exec/function_group_state.h"
Scope:	`class ara::exec::FunctionGroupState`
Syntax:	`~FunctionGroupState () noexcept;`
Exception Safety:	exception safe
Thread Safety:	implementation defined
Description:	Destructor of the FunctionGroupState instance.

⌋

8.2.3.7 FunctionGroupState::operator==

[SWS_EM_02273] Definition of API function ara::exec::FunctionGroupState::operator==

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`bool operator== (const FunctionGroupState &other) const noexcept;`	
Parameters (in):	other	FunctionGroupState instance to compare this one with.
Return value:	bool	true in case both FunctionGroupStates are representing exactly the same meta-model element, false otherwise.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	eq operator to compare with other FunctionGroupState instance.	

⌋

8.2.3.8 FunctionGroupState::operator!=

[SWS_EM_02274] Definition of API function ara::exec::FunctionGroupState::operator!=

Upstream requirements: RS_EM_00101

⌈

Kind:	function	
Header file:	#include "ara/exec/function_group_state.h"	
Scope:	`class ara::exec::FunctionGroupState`	
Syntax:	`bool operator!= (const FunctionGroupState &other) const noexcept;`	
Parameters (in):	other	FunctionGroupState instance to compare this one with.
Return value:	bool	false in case both FunctionGroupStates are representing exactly the same meta-model element, true otherwise.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	uneq operator to compare with other FunctionGroupState instance.	

⌋

8.2.4 StateClient class

Class used to perform `Function Group` state management operation needed during lifetime of a `Machine`. `State Management` during its own lifetime will need to start and stop software, that is intended to run on a `Machine` managed by it. This can be achieved by performing state transition of a `Function Group` to which required software is assigned. Integrator will assign software to run in a particular state (of `Function Group`) and `State Management` can start it, by requesting `Execution Management` to perform state transition (of this `Function Group`) to the mentioned state. `Execution Management` will then start mentioned software and report transition result back to `State Management`. Please note that stopping software can be done in similar way (i.e. `Function Group` state transition, to a state in which software is not configured to be run).

[SWS_EM_02275] Definition of API class ara::exec::StateClient

Upstream requirements: RS_EM_00101, RS_AP_00154, RS_AP_00150

⌈

Kind:	class
Header file:	#include "ara/exec/state_client.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	StateClient
Syntax:	`class StateClient final {...};`
Description:	StateClient is an interface of Execution Management that is used by State Management to request transitions between Function Group States or to perform other related operations.
Notes:	StateClient opens communication channel to Execution Management (e.g. POSIX FIFO). Each Process that intends to perform state management, should create an instance of this class and it should have rights to use it. To eventually implement the Named Constructor Idiom, the developer may either make the default constructor private or delete it and define a non-default constructor.

⌋

8.2.4.1 StateClient::StateClient

[SWS_EM_02561] Definition of API function ara::exec::StateClient::StateClient

Upstream requirements: RS_EM_00101, RS_AP_00114, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00151

⌈

Kind:	function	
Header file:	#include "ara/exec/state_client.h"	
Scope:	`class ara::exec::StateClient`	
Syntax:	`StateClient (std::function< void(const ara::exec::ExecutionErrorEvent &)> undefinedStateCallback) noexcept(false);`	
Parameters (in):	undefinedStateCallback	callback to be invoked by StateClient library if a FunctionGroup changes its state unexpectedly to an Undefined Function Group State, i.e. without previous request by SetState(). The affected FunctionGroup and ExecutionError is provided as an argument to the callback in form of ExecutionErrorEvent Lifetime: it is expected that undefinedStateCallback remains callable, during entire lifetime of the StateClient instance. This is especially important if, undefinedStateCallback is bound to an instance of a class (e.g. using std::bind).
Exception Safety:	not exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kNo Communication	--
		communication error occurred
	ara::exec::ExecErrc::k InvalidArgument	--
		Given terminationHandler doesn't contain a callable function.
Description:	Regular constructor for StateClient.	

⌋

8.2.4.2 StateClient::Create

[SWS_EM_02276] Definition of API function ara::exec::StateClient::Create

Upstream requirements: RS_EM_00101, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00139, RS_AP_00144

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`static ara::core::Result< StateClient > Create (std::function< void(const ara::exec::ExecutionErrorEvent &)> undefinedStateCallback) noexcept;`

▽

△

Parameters (in):	undefinedStateCallback	callback to be invoked by StateClient library if a FunctionGroup changes its state unexpectedly to an Undefined Function Group State, i.e. without previous request by SetState(). The affected FunctionGroup and ExecutionError is provided as an argument to the callback in form of ExecutionErrorEvent. Lifetime: it is expected that undefinedStateCallback remains callable, during entire lifetime of the StateClient instance. This is especially important if, undefinedStateCallback is bound to an instance of a class (e.g. using std::bind).
Return value:	ara::core::Result< State Client >	a result that contains either a StateClient object or an error.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kNo Communication	--
		communication error occurred
	ara::exec::ExecErrc::k InvalidArgument	--
		Given terminationHandler doesn't contain a callable function.
Description:	Named constructor for StateClient.	
Notes:	This named constructor may call a private constructor defined by the developer.	

⌋

8.2.4.3 StateClient::~StateClient

[SWS_EM_02277] Definition of API function ara::exec::StateClient::~StateClient

Upstream requirements: RS_AP_00134, RS_AP_00145, RS_EM_00101

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`~StateClient () noexcept;`
Exception Safety:	exception safe
Thread Safety:	implementation defined
Description:	noexcept destructor

⌋

8.2.4.4 StateClient::StateClient (deleted Copy Constructor)

[SWS_EM_02565] Definition of API function ara::exec::StateClient::StateClient

Upstream requirements: RS_AP_00145, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`StateClient (const StateClient &)=delete;`
Description:	Suppress default copy construction for StateClient.

⌋

8.2.4.5 StateClient::operator= (deleted Copy assignment operator)

[SWS_EM_02568] Definition of API function ara::exec::StateClient::operator=

Upstream requirements: RS_AP_00145, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`StateClient & operator= (const StateClient &)=delete;`
Description:	Suppress default copy assignment for StateClient.

⌋

8.2.4.6 StateClient::StateClient (use of default move constructor)

[SWS_EM_02566] Definition of API function ara::exec::StateClient::StateClient

Upstream requirements: RS_AP_00145, RS_AP_00151, RS_EM_00103

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`StateClient (StateClient &&rval) noexcept;`

▽

△

Parameters (in):	rval	reference to move
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Intentional use of default move constructor for StateClient.	

⌋

8.2.4.7 StateClient::operator= (use of default move assignment)

[SWS_EM_02567] Definition of API function ara::exec::StateClient::operator=

Upstream requirements: RS_AP_00145, RS_AP_00151, RS_EM_00103

⌈

Kind:	function	
Header file:	#include "ara/exec/state_client.h"	
Scope:	`class ara::exec::StateClient`	
Syntax:	`StateClient & operator= (StateClient &&rval) noexcept;`	
Parameters (in):	rval	reference to move
Return value:	StateClient &	the new reference
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Intentional use of default move assignment for StateClient.	

⌋

8.2.4.8 StateClient::SetState

[SWS_EM_02278] Definition of API function ara::exec::StateClient::SetState

Upstream requirements: RS_EM_00101, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00138, RS_AP_00128

⌈

Kind:	function
Header file:	#include "ara/exec/state_client.h"
Scope:	`class ara::exec::StateClient`
Syntax:	`ara::core::Future< void > SetState (const FunctionGroupState &state) const noexcept;`

▽

△

Parameters (in):	state	representing meta-model definition of a state inside a specific Function Group. Execution Management will perform state transition from the current state to the state identified by this parameter.
Return value:	ara::core::Future< void >	void if requested transition is successful, otherwise it returns Exec ErrorDomain error.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kOperationCanceled	--
		if transition to the requested Function Group state was cancelled by a newer request
	ara::exec::ExecErrc::kOperationFailed	--
		if transition to the requested Function Group state failed
	ara::exec::ExecErrc::kNoCommunication	--
		if StateClient can't communicate with Execution Management (e.g. IPC link is down)
	ara::exec::ExecErrc::kInvalidTransition	--
		if transition to the requested state is prohibited (e.g. Off state for MachineFG) or the requested Function Group State is invalid (e.g. does not exist anymore after a software update)
	ara::exec::ExecErrc::kIntegrityOrAuthenticityCheckFailed	--
		if an integrity or authenticity check failed during state transition.
	ara::exec::ExecErrc::kUnexpectedTermination	--
		One of the processes terminated in an unexpected way during the state transition.
	ara::exec::ExecErrc::kInvalidMetaModelIdentifier	--
		The given Function Group State couldn't be found in the Processed Manifest or Process does not have a mapping to the Function Group of the requested Function Group State.
Description:	Method to request state transition for a single Function Group. This method will request Execution Management to perform state transition and return immediately. Returned ara::core::Future can be used to determine result of requested transition.	

⌋

Asynchronous nature of `ara::exec::StateClient::SetState` makes the returned `ara::core::Future` dependable on lifetime of the instance from which it was received. It is expected that once state change request is received by `Execution Management`, it will be processed independently of lifetime of the instance from which it was requested.

Please note that the qualified short names representing FunctionGroups and FunctionGroupStates could be quite long. They can be replaced by implementation specific data types to speed up any checks that have to be performed by the SetState method. This is enabled by the FunctionGroupState data-type.

8.2.4.9 StateClient::GetInitialMachineStateTransitionResult

[SWS_EM_02279] Definition of API function ara::exec::StateClient::GetInitialMachineStateTransitionResult

Upstream requirements: RS_EM_00101, RS_AP_00119, RS_AP_00120, RS_AP_00138, RS_AP_00128

⌈

Kind:	function	
Header file:	#include "ara/exec/state_client.h"	
Scope:	`class ara::exec::StateClient`	
Syntax:	`ara::core::Future< void > GetInitialMachineStateTransitionResult () const noexcept;`	
Return value:	ara::core::Future< void >	void if requested transition is successful, otherwise it returns Exec ErrorDomain error.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::kOperationCanceled	--
		StateManagement may decide to cancel SWS_EM_01023 transition and start specific startup sequence. This could happen for number of reasons and one of them could be interrupted Machine update sequence.
	ara::exec::ExecErrc::kOperationFailed	--
		if transition to the requested Function Group state failed
	ara::exec::ExecErrc::kNoCommunication	--
		if StateClient can't communicate with Execution Management (e.g. IPC link is down)
	ara::exec::ExecErrc::kInvalidMetaModelIdentifier	--
		Process does not have a mapping to MachineFG.
Description:	Method to retrieve result of Machine State initial transition to Startup state.	
Notes:	This method allows State Management to retrieve the result of a transition specified by SWS_EM_01023 and SWS_EM_02241. Please note that this transition happens once per machine life cycle, thus the result delivered by this method shall not change (unless machine is started again).	

⌋

Please note that concerns about returned `ara::core::Future` from `ara::exec::StateClient::SetState` apply for `ara::exec::StateClient::GetInitialMachineStateTransitionResult`.

8.2.4.10 StateClient::GetExecutionError

[SWS_EM_02542] Definition of API function ara::exec::StateClient::GetExecutionError

Upstream requirements: RS_EM_00101, RS_AP_00120, RS_AP_00121, RS_AP_00128, RS_AP_00139

⌈

Kind:	function	
Header file:	#include "ara/exec/state_client.h"	
Scope:	`class ara::exec::StateClient`	
Syntax:	`ara::core::Result< ara::exec::ExecutionErrorEvent > GetExecutionError (const ara::exec::FunctionGroupState &functionGroupState) noexcept;`	
Parameters (in):	functionGroupState	Function Group State of interest.
Return value:	ara::core::Result< ara::exec::Execution ErrorEvent >	The execution error which changed the Function Group of the given Function Group State to an Undefined Function Group State.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Errors:	ara::exec::ExecErrc::k OperationFailed	--
		The Function Group of the given Function Group State is not in an Undefined Function Group State.
	ara::exec::ExecErrc::kNo Communication	--
		if StateClient can't communicate with Execution Management (e.g. IPC link is down)
	ara::exec::ExecErrc::k InvalidMetaModel Identifier	--
		The given Function Group State couldn't be found in the Processed Manifest or Process does not have a mapping to the Function Group of the given Function Group State.
Description:	Returns the execution error which changed the Function Group of the given Function Group State to an Undefined Function Group State. This function will return with error and will not return an ExecutionErrorEvent object, if the Function Group is in a defined Function Group state again.	

⌋

[SWS_EM_02543] Default value for ExecutionError

Upstream requirements: RS_EM_00101

⌈In case of `Unexpected Termination` or `Unexpected Self-termination` of a `Modelled Process` which does not have an `executionError` configured, `Execution Management` shall report the `ExecutionError` value 1.⌋

8.3 Log and Trace Messages

[SWS_EM_02569] LogMessage ProcessCreated

Status: DRAFT
Upstream requirements: RS_EM_00152, RS_AP_00156

⌈

Dlt-Message	ProcessCreated		
Description	Message that is sent by the Execution Management right after the Execution Management successfully created a process.		
MessageId	0x80004001		
MessageType Info	DLT_TRACE_VFB		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
processId	OS specific PID which has been assigned to the process.	uint32	NoUnit
processName	Shortname of the Process element which has been started.	uint8 [encoding UTF-8]	NoUnit
is_created	is created	predefined text	

⌋

[SWS_EM_02570] LogMessage ProcessKRunningReceived

Status: DRAFT
Upstream requirements: RS_EM_00152, RS_AP_00156

⌈

Dlt-Message	ProcessKRunningReceived		
Description	Message that is sent by the Execution Management when the Execution Management received a k Running from ara::exec::ExecutionClient::ReportExecutionState.		
MessageId	0x80004002		
MessageType Info	DLT_TRACE_VFB		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
processId	OS specific PID which has been assigned to the process.	uint32	NoUnit
received_k Running	received kRunning	predefined text	

⌋

[SWS_EM_02571] LogMessage ProcessTerminationRequest

Status: DRAFT
Upstream requirements: RS_EM_00152, RS_AP_00156

⌈

Dlt-Message	ProcessTerminationRequest		
Description	Message that is sent by the Execution Management when the Execution Management requested to terminate a process.		
MessageId	0x80004003		
MessageType Info	DLT_TRACE_VFB		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
execution_ management_ requested_to_ terminate	Execution Management requested to terminate	predefined text	
processId	OS specific PID which has been assigned to the process.	uint32	NoUnit

⌋

[SWS_EM_02572] LogMessage ProcessTerminated

Status: DRAFT
Upstream requirements: RS_EM_00152, RS_AP_00156

⌈

Dlt-Message	ProcessTerminated		
Description	Message that is sent by the Execution Management when the Execution Management received an EXIT_SUCCESS or an unexpected termination from the process.		
MessageId	0x80004004		
MessageType Info	DLT_TRACE_VFB		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
processId	OS specific PID which has been assigned to the process.	uint32	NoUnit
is_terminated	is terminated	predefined text	

⌋

8.4 Errors

The Execution Management cluster implements an error handling based on ara::core::Result. The errors supported by the Execution Management cluster are listed in Section 8.4.1.

8.4.1 Execution Management error codes

[SWS_EM_02281] Definition of API enum ara::exec::ExecErrc

Upstream requirements: RS_AP_00130, RS_AP_00122, RS_AP_00127, RS_AP_00154, RS_AP_00125, RS_AP_00142, RS_AP_00129, RS_AP_00149

⌈

Kind:	enumeration	
Header file:	#include "ara/exec/exec_error_domain.h"	
Forwarding header file:	#include "ara/exec/exec_fwd.h"	
Scope:	namespace ara::exec	
Symbol:	ExecErrc	
Underlying type:	ara::core::ErrorDomain::CodeType	
Syntax:	enum class ExecErrc : ara::core::ErrorDomain::CodeType {...};	
Values:	kNoCommunication= 3	Communication error occurred, e.g. request cannot be send or result cannot be retrieved
	kInvalidMetaModel Identifier= 4	Wrong meta model identifier passed to a function
	kOperationCanceled= 5	Transition to the requested Function Group state was canceled by a newer request
	kOperationFailed= 6	Requested operation could not be performed, e.g. Function Group state transition cannot be finished because kRunning was not reported on time
	kInvalidTransition= 9	Invalid transition (e.g. Process attempted to report kRunning, when it was already in a Running Process State)
	kIntegrityOrAuthenticity CheckFailed= 14	Integrity or authenticity check for a Process to be spawned in the requested Function Group state failed
	kUnexpected Termination= 15	Unexpected Termination during a Function Group State transition occurred
	kInvalidArgument= 16	Passed argument doesn't appear to be valid.
Description:	Defines an enumeration class for the Execution Management error codes.	

⌋

Please note Execution Management intentionally does not consider, a wrong meta model identifier passed to a function, as a violation. Violation handling as required by [RS_AP_00142] leads to Unexpected Termination, which would be reported to and handled by State Management. Note that for APIs intended to be used by State Management itself, this error handling strategy is not considered to contribute

to robust system design (State Management would be terminated by [RS_AP_-00142]). Therefore ara::exec APIs which are intended to be used by State Management return kInvalidMetaModelIdentifier instead, as this allows implementation of fallback strategies within State Management.

If a kNoCommunication occurs at Execution Management or State Management level (or communication between both), the system may need to enter an Unrecoverable State. The internal mechanisms can detect this issue in the moment that a communication attempt occurs. This is not the case for applications that are using the interface of the ExecutionClient or StateClient - they eventually get a communication error in the moment of the API use, i.e. possibly after the actual error occurrence. However, this is not problematic as the recovery options are limited from a client side viewpoint.

8.4.2 ExecException class

[SWS_EM_02282] Definition of API class ara::exec::ExecException

Upstream requirements: RS_AP_00130, RS_AP_00122, RS_AP_00127, RS_AP_00154, RS_AP_00150, RS_AP_00140

⌈

Kind:	class
Header file:	#include "ara/exec/exec_error_domain.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	ExecException
Base class:	ara::core::Exception
Syntax:	`class ExecException final : public ara::core::Exception {...};`
Description:	Defines a class for exceptions to be thrown by the Execution Management.

⌋

8.4.2.1 ExecException::ExecException

[SWS_EM_02283] Definition of API function ara::exec::ExecException::ExecException

Upstream requirements: RS_AP_00120, RS_AP_00121, RS_AP_00130

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`class ara::exec::ExecException`	
Syntax:	`explicit ExecException (ara::core::ErrorCode errorCode) noexcept;`	
Parameters (in):	errorCode	The error code.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Constructs a new ExecException object containing an error code.	

⌋

8.4.3 GetExecErrorDomain function

[SWS_EM_02290] Definition of API function ara::exec::GetExecErrorDomain

Upstream requirements: RS_AP_00120, RS_AP_00130, RS_AP_00154

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`namespace ara::exec`	
Syntax:	`const ara::core::ErrorDomain & GetExecErrorDomain () noexcept;`	
Return value:	const ara::core::Error Domain &	Return a reference to the global ExecErrorDomain object.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Returns a reference to the global ExecErrorDomain object.	

⌋

8.4.4 MakeErrorCode function

[SWS_EM_02291] Definition of API function ara::exec::MakeErrorCode

Upstream requirements: RS_AP_00120, RS_AP_00121, RS_AP_00130, RS_AP_00154

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`namespace ara::exec`	
Syntax:	`ara::core::ErrorCode MakeErrorCode (ara::exec::ExecErrc code, ara::core::ErrorDomain::SupportDataType data) noexcept;`	
Parameters (in):	code	Error code number.
	data	Vendor defined data associated with the error.
Return value:	ara::core::ErrorCode	An ErrorCode object.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Creates an instance of ErrorCode.	

⌋

8.4.5 ExecErrorDomain class

The error handling requires an `ara::core::ErrorDomain`, which can be used to check the errors returned via `ara::core::Result`.

[SWS_EM_02284] Definition of API class ara::exec::ExecErrorDomain

Upstream requirements: RS_AP_00130, RS_AP_00122, RS_AP_00127, RS_AP_00154, RS_AP_00150

⌈

Kind:	class
Header file:	#include "ara/exec/exec_error_domain.h"
Forwarding header file:	#include "ara/exec/exec_fwd.h"
Scope:	`namespace ara::exec`
Symbol:	ExecErrorDomain
Base class:	ara::core::ErrorDomain
Syntax:	`class ExecErrorDomain final : public ara::core::ErrorDomain {...};`
Unique ID:	As per `ara::exec::ExecErrorDomain` in [SWS_CORE_90023]
Description:	Defines a class representing the Execution Management error domain.

⌋

8.4.5.1 ExecErrorDomain::ExecErrorDomain

[SWS_EM_02286] Definition of API function ara::exec::ExecErrorDomain::ExecErrorDomain

Upstream requirements: RS_AP_00120, RS_AP_00130

⌈

Kind:	function
Header file:	#include "ara/exec/exec_error_domain.h"
Scope:	`class ara::exec::ExecErrorDomain`
Syntax:	`ExecErrorDomain () noexcept;`
Exception Safety:	exception safe
Thread Safety:	implementation defined
Description:	Constructs a new ExecErrorDomain object.

⌋

8.4.5.2 ExecErrorDomain::Name

[SWS_EM_02287] Definition of API function ara::exec::ExecErrorDomain::Name

Upstream requirements: RS_AP_00120, RS_AP_00130

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`class ara::exec::ExecErrorDomain`	
Syntax:	`const char * Name () const noexcept override;`	
Return value:	const char *	"Exec".
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Returns a string constant associated with ExecErrorDomain.	

⌋

8.4.5.3 ExecErrorDomain::Message

[SWS_EM_02288] Definition of API function ara::exec::ExecErrorDomain::Message

Upstream requirements: RS_AP_00120, RS_AP_00121, RS_AP_00130

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`class ara::exec::ExecErrorDomain`	
Syntax:	`const char * Message (CodeType errorCode) const noexcept override;`	
Parameters (in):	errorCode	The error code number.
Return value:	const char *	The message associated with the error code.
Exception Safety:	exception safe	
Thread Safety:	implementation defined	
Description:	Returns the message associated with errorCode.	

⌋

8.4.5.4 ExecErrorDomain::ThrowAsException

[SWS_EM_02289] Definition of API function ara::exec::ExecErrorDomain::ThrowAsException

Upstream requirements: RS_AP_00120, RS_AP_00121, RS_AP_00130

⌈

Kind:	function	
Header file:	#include "ara/exec/exec_error_domain.h"	
Scope:	`class ara::exec::ExecErrorDomain`	
Syntax:	`void ThrowAsException (const ara::core::ErrorCode &errorCode) const noexcept(false) override;`	
Parameters (in):	errorCode	The error to throw.
Return value:	None	
Exception Safety:	not exception safe	
Thread Safety:	implementation defined	
Description:	Creates a new instance of ExecException from errorCode and throws it as a C++ exception. As per [SWS_CORE_10304], this function does not participate in overload resolution when C++ exceptions are disabled in the compiler toolchain.	

⌋

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.

Class	**Executable**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure			
Note	This meta-class represents an executable program. **Tags:** atp.recommendedPackage=Executables			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
buildType	BuildTypeEnum	0..1	attr	This attribute describes the buildType of a module and/or platform implementation.
implementation Props	Executable ImplementationProps	*	aggr	This aggregation contains the collection of implementation-specific properties necessary to properly build the enclosing Executable.
minimumTimer Granularity	TimeValue	0..1	attr	This attribute describes the minimum timer resolution (TimeValue of one tick) that is required by the Executable.
reporting Behavior	ExecutionState ReportingBehavior Enum	0..1	attr	this attribute controls the execution state reporting behavior of the enclosing Executable.
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.
traceSwitch Configuration	TraceSwitch Configuration	*	aggr	Configuration of the MsgId based trace switch **Tags:** atp.Status=draft
version	StrongRevisionLabel String	0..1	attr	Version of the executable.

Table A.1: Executable

Class	**ExecutionDependency**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This element defines a ProcessState in which a dependent process needs to be before the process that aggregates the ExecutionDependency element can be started.			
Base	*ARObject*			
Aggregated by	StateDependentStartupConfig.executionDependency			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
processState	ModeDeclaration	0..1	iref	This represent the applicable modeDeclaration that represents an ProcessState. **InstanceRef implemented by:** ModeInProcessInstance Ref

Table A.2: ExecutionDependency

Enumeration	**ExecutionStateReportingBehaviorEnum**
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure
Note	This enumeration provides options for controlling of how an Executable reports its execution state to the Execution Management
Aggregated by	Executable.reportingBehavior
Literal	***Description***
doesNotReport ExecutionState	The Executable shall not report its execution state to the Execution Management. **Tags:** atp.EnumerationLiteralIndex=1
reportsExecution State	The Executable shall report its execution state to the Execution Management. **Tags:** atp.EnumerationLiteralIndex=0

Table A.3: ExecutionStateReportingBehaviorEnum

Class	**FunctionGroupPortMapping**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::ExecutionManagement			
Note	This class is used to associate a PortPrototype typed by a StateClientInterface with the actual function group to which the state changes communicated over the PortPrototype shall apply. **Tags:** atp.Status=draft atp.recommendedPackage=FunctionGroupPortMappings			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
functionGroup	ModeDeclarationGroup Prototype	0..1	ref	This reference identifies the applicable function group for which the state change shall be executed. **Tags:** atp.Status=draft
process	Process	0..1	ref	This reference identifies the Process of the state client **Tags:** atp.Status=draft
rPortPrototype InExecutable	RPortPrototype	0..1	iref	This reference identifies the applicable PortPrototype for the function group state change. **Stereotypes:** atpUriDef **Tags:** atp.Status=draft **InstanceRef implemented by:** RPortPrototypeIn ExecutableInstanceRef

Table A.4: FunctionGroupPortMapping

Class	**Machine**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::SubSystemDesign::MachineManifest			
Note	Machine that represents an Adaptive Autosar Software Stack. **Tags:** atp.recommendedPackage=Machines			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeployment Element*, *UploadablePackageElement*			
Aggregated by	ARPackage.element, *AtpClassifier*.atpFeature			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
default Application Timeout	EnterExitTimeout	0..1	aggr	This aggregation defines a default timeout in the context of a given Machine with respect to the launching and termination of applications.

△

Class	**Machine**			
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
machineDesign	MachineDesign	0..1	ref	Reference to the MachineDesign this Machine is implementing.
module Instantiation	AdaptiveModule Instantiation	*	aggr	Configuration of Adaptive Autosar module instances that are running on the machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=moduleInstantiation.shortName
processor	Processor	*	aggr	This represents the collection of processors owned by the enclosing machine.
secure Communication Deployment	SecureCommunication Deployment	*	aggr	Target-configuration of secure communication protocol configuration settings to crypto module entities. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=secureCommunication Deployment.shortName
trustedPlatform Executable LaunchBehavior	TrustedPlatform ExecutableLaunch BehaviorEnum	0..1	attr	This attribute controls the behavior of how authentication affects the ability to launch for each Executable.

Table A.5: 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.			
Base	*ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, ModeDeclarationGroup.modeDeclaration			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
value	PositiveInteger	0..1	attr	The RTE shall take the value of this attribute for generating the source code representation of this Mode Declaration.

Table A.6: ModeDeclaration

Class	**ModeDeclarationGroup**			
Package	M2::AUTOSARTemplates::CommonStructure::ModeDeclaration			
Note	A collection of Mode Declarations. Also, the initial mode is explicitly identified. **Tags:** atp.recommendedPackage=ModeDeclarationGroups			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
initialMode	ModeDeclaration	0..1	ref	The initial mode of the ModeDeclarationGroup. This mode is active before any mode switches occurred.

▽

△

Class	**ModeDeclarationGroup**			
mode Declaration	ModeDeclaration	*	aggr	The ModeDeclarations collected in this ModeDeclaration Group. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=modeDeclaration.shortName, mode Declaration.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime

Table A.7: ModeDeclarationGroup

Class	**ModeDeclarationGroupPrototype**			
Package	M2::AUTOSARTemplates::CommonStructure::ModeDeclaration			
Note	The ModeDeclarationGroupPrototype specifies a set of Modes (ModeDeclarationGroup) which is provided or required in the given context.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, BswModuleDescription.providedModeGroup, BswModuleDescription.requiredModeGroup, FirewallStateSwitchInterface.firewallStateMachine, FunctionGroupSet.functionGroup, ModeSwitchInterface.modeGroup, Process.processStateMachine, StateManagementStateNotification.stateMachine			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
type	ModeDeclarationGroup	0..1	tref	The "collection of ModeDeclarations" (= ModeDeclarationGroup) supported by a component **Stereotypes:** isOfType

Table A.8: ModeDeclarationGroupPrototype

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class provides information required to execute the referenced `Executable`. **Tags:** atp.recommendedPackage=Processes			
Base	*ARElement*, *ARObject*, *AbstractExecutionContext*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
design	ProcessDesign	0..1	ref	This reference represents the identification of the design-time representation for the Process that owns the reference.
executable	Executable	*	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef
functionClusterAffiliation	String	0..1	attr	This attribute specifies which functional cluster the Process is affiliated with.
numberOfRestartAttempts	PositiveInteger	0..1	attr	This attribute defines how often a process shall be restarted if the start fails. numberOfRestartAttempts = "0" OR Attribute not existing, start once numberOfRestartAttempts = "1", start a second time
preMapping	Boolean	0..1	attr	This attribute describes whether the executable is preloaded into the memory.

▽

△

Class	Process			
processState Machine	ModeDeclarationGroup Prototype	0..1	aggr	Set of Process States that are defined for the process. This attribute is used to support the modeling of execution dependencies that utilize the condition of process state. Please note that the process states may not be modeled arbitrarily at any stage of the AUTOSAR workflow because the supported states are standardized in the context of the SWS Execution Management [17].
stateDependent StartupConfig	StateDependentStartup Config	*	aggr	Applicable startup configurations.

Table A.9: Process

Class	ProcessArgument			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class has the ability to define command line arguments for processing by the Main function.			
Base	*ARObject*			
Aggregated by	StartupConfig.processArgument			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
argument	String	0..1	attr	This represents one command-line argument to be processed by the executable software.

Table A.10: ProcessArgument

Class	ProcessToMachineMapping			
Package	M2::AUTOSARTemplates::AdaptivePlatform::SubSystemDesign::MachineManifest			
Note	This meta-class has the ability to associate a Process with a Machine. This relation involves the definition of further properties, e.g. timeouts.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	ProcessToMachineMappingSet.processToMachineMapping			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
design	ProcessDesignTo MachineDesignMapping	0..1	ref	This reference represents the identification of the design-time representation for the ProcessToMachine Mapping that owns the reference.
machine	Machine	0..1	ref	This reference identifies the Machine in the context of the ProcessToMachineMapping.
nonOsModule Instantiation	NonOsModule Instantiation	0..1	ref	This supports the optional case that the process represents a platform module.
persistency CentralStorage URI	UriString	0..1	attr	This attribute identifies a central place for the mapped Process to store the list of available storages and version information.
process	Process	0..1	ref	This reference identifies the Process in the context of the ProcessToMachineMapping.
shallNotRunOn	ProcessorCore	*	ref	This reference indicates a collection of cores onto which the mapped process shall not be executing.
shallRunOn	ProcessorCore	*	ref	This reference indicates a collection of cores onto which the mapped process shall be executing.

Table A.11: ProcessToMachineMapping

Class	***Referrable*** (abstract)			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable			
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base	*ARObject*			
Subclasses	*AtpDefinition*, BswDistinguishedPartition, *BswModuleCallPoint*, BswModuleClientServerEntry, BswVariableAccess, CouplingPortTrafficClassAssignment, *CppImplementationDataTypeContextTarget*, *DiagnosticEnvModeElement*, EthernetPriorityRegeneration, ExclusiveAreaNestingOrder, *HwDescriptionEntity*, *ImplementationProps*, ModeTransition, *MultilanguageReferrable*, NmNetworkHandle, PncMappingIdent, *SingleLanguageReferrable*, SoConIPduIdentifier, SocketConnectionBundle, SomeipRequiredEventGroup, TimeSyncServerConfiguration, TpConnectionIdent			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Stereotypes:** atpIdentityContributor **Tags:** xml.enforceMinMultiplicity=true xml.sequenceOffset=-100
shortNameFragment	ShortNameFragment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:** xml.sequenceOffset=-90

Table A.12: Referrable

Class	**ResourceConsumption**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption			
Note	Description of consumed resources by one implementation of a software.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	EcuResourceEstimation.bswResourceEstimation, EcuResourceEstimation.rteResourceEstimation, *Implementation*.resourceConsumption, StateDependentStartupConfig.resourceConsumption			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
memoryUsage	MemoryUsage	*	aggr	Collection of the memory allocated by the owner. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=memoryUsage.shortName

Table A.13: ResourceConsumption

Class	**ResourceGroup**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::AdaptiveModuleImplementation			
Note	This meta-class represents a resource group that limits the resource usage of a collection of processes.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	OsModuleInstantiation.resourceGroup			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
cpuUsage	PositiveInteger	0..1	attr	CPU resource limit in percentage of the total CPU capacity on the machine.
memUsage	PositiveInteger	0..1	attr	Memory limit in bytes.

Table A.14: ResourceGroup

Class	**SoftwareCluster**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::SoftwareDistribution			
Note	This meta-class represents the ability to define an uploadable software-package, i.e. the SoftwareCluster shall contain all software and configuration for a given purpose. **Tags:** atp.recommendedPackage=SoftwareClusters			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
artifact Checksum	ArtifactChecksum	*	aggr	This aggregation carries the checksums for artifacts contained in the enclosing SoftwareCluster. Please note that the value of these checksums is only applicable at the time of configuration. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=artifactChecksum.shortName, artifactChecksum.uri
artifactLocator	ArtifactLocator	*	aggr	This aggregation represents the artifact locations that are relevant in the context of the enclosing SoftwareCluster
claimed FunctionGroup	ModeDeclarationGroupPrototype	*	ref	Each SoftwareCluster can reserve the usage of a given functionGroup such that no other SoftwareCluster is allowed to use it
conflictsTo	SoftwareClusterDependencyFormula	0..1	aggr	This aggregation handles conflicts. If it yields true then the SoftwareCluster shall not be installed. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=conflictsTo
contained ARElement	ARElement	*	ref	This reference represents the collection of model elements that cannot derive from UploadablePackageElement and that contribute to the completeness of the definition of the SoftwareCluster. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=containedARElement
containedFibexElement	FibexElement	*	ref	This allows for referencing FibexElements that need to be considered in the context of a SoftwareCluster.
contained PackageElement	UploadablePackageElement	*	ref	This reference identifies model elements that are required to complete the manifest content. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=containedPackageElement
contained Process	Process	*	ref	This reference represent the processes contained in the enclosing SoftwareCluster.
dependsOn	SoftwareClusterDependencyFormula	0..1	aggr	This aggregation can be taken to identify a dependency for the enclosing SoftwareCluster. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=dependsOn
design	SoftwareClusterDesign	*	ref	This reference represents the identification of all SoftwareClusterDesigns applicable for the enclosing SoftwareCluster. **Stereotypes:** atpUriDef
diagnosticDeploymentProps	SoftwareClusterDiagnosticDeploymentProps	0..1	ref	This reference identifies the applicable SoftwareClusterDiagnosticDeploymentProps that are applicable for the referencing SoftwareCluster.
installationBehavior	SoftwareClusterInstallationBehaviorEnum	0..1	attr	This attribute controls the behavior of the SoftwareCluster in terms of installation.

▽

△

Class	**SoftwareCluster**			
license	Documentation	*	ref	This attribute allows for the inclusion of the full text of a license of the enclosing SoftwareCluster. In many cases open source licenses require the inclusion of the full license text to any software that is released under the respective license.
module Instantiation	AdaptiveModule Instantiation	*	ref	This reference identifies AdaptiveModuleInstantiations that need to be included with the SoftwareCluster in order to establish infrastructure required for the installation of the SoftwareCluster. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=moduleInstantiation
releaseNotes	Documentation	0..1	ref	This attribute allows for the explanations of changes since the previous version. The list of changes might require the creation of multiple paragraphs of test.
typeApproval	String	0..1	attr	This attribute carries the homologation information that may be specific for a given country.
vendorId	PositiveInteger	0..1	attr	Vendor ID of this Implementation according to the AUTOSAR vendor list.
vendor Signature	CryptoService Certificate	0..1	ref	This reference identifies the certificate that represents the vendor's signature.
version	StrongRevisionLabel String	0..1	attr	This attribute can be used to describe a version information for the enclosing SoftwareCluster.

Table A.15: SoftwareCluster

Class	**StartupConfig**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class represents a reusable startup configuration for processes.. **Tags:** atp.recommendedPackage=StartupConfigs			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***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.
executionError	ProcessExecutionError	0..1	ref	this reference is used to identify the applicable execution error
permissionTo CreateChild Process	Boolean	0..1	attr	This attribute defines if Process is permitted to create child Processes. When setting this parameter to true two things should be kept in mind: 1) safety and security implication of this configuration, 2) the fact that Process will assume management responsibilities for child Processes (i.e. it will be responsible for terminating Processes that it creates).
process Argument (ordered)	ProcessArgument	*	aggr	This aggregation represents the collection of command-line arguments applicable to the enclosing StartupConfig.
scheduling Policy	String	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.
termination Behavior	TerminationBehavior Enum	0..1	attr	This attribute defines the termination behavior of the Process.

▽

△

Class	StartupConfig			
timeout	EnterExitTimeout	0..1	aggr	This aggregation can be used to specify the timeouts for launching and terminating the process depending on the StartupConfig.

Table A.16: StartupConfig

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.			
Base	*ARObject*			
Aggregated by	Process.stateDependentStartupConfig			
Attribute	**Type**	**Mult.**	**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.
functionGroup State	ModeDeclaration	*	iref	This represent the applicable functionGroupMode. **InstanceRef implemented by:** FunctionGroupStateIn FunctionGroupSetInstanceRef
resource Consumption	ResourceConsumption	0..1	aggr	This aggregation provides the ability to define resource consumption boundaries on a per-process-startup-config basis.
resourceGroup	ResourceGroup	0..1	ref	Reference to an applicable resource group.
startupConfig	StartupConfig	0..1	ref	Reference to a reusable startup configuration with startup parameters.

Table A.17: 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.			
Base	*ARObject*			
Aggregated by	*AbstractServiceInstance*.capabilityRecord, Machine.environmentVariable, ProvidedSomeipService Instance.capabilityRecord, RequiredSomeipServiceInstance.capabilityRecord, SdClientConfig.capability Record, SdServerConfig.capabilityRecord, StartupConfig.environmentVariable			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
key	String	0..1	attr	Defines a key.
sequenceOffset	Integer	0..1	attr	The sequenceOffset attribute supports the use case where TagWithOptionalValue is aggregated as splitable. If multiple aggregations define the same value of attribute key then the order in which the value collection is merged might be significant. As an example consider the modeling of the $PATH environment variable by means of a meta class TagWithOptionalValue. The sequenceOffset describes the relative position of each contribution in the concatenated value. The contributions are sorted in increasing integer order.
value	String	0..1	attr	Defines the corresponding value.

Table A.18: TagWithOptionalValue

Enumeration	**TerminationBehaviorEnum**
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest
Note	This enumeration provides options for controlling of how a Process terminates.
Aggregated by	StartupConfig.terminationBehavior
Literal	***Description***
processIsNotSelf Terminating	The Process terminates only on request from Execution Management. **Tags:** atp.EnumerationLiteralIndex=0
processIsSelf Terminating	The Process is allowed to terminate without request from Execution Management. **Tags:** atp.EnumerationLiteralIndex=1

Table A.19: TerminationBehaviorEnum

Enumeration	**TrustedPlatformExecutableLaunchBehaviorEnum**
Package	M2::AUTOSARTemplates::AdaptivePlatform::SubSystemDesign::MachineManifest
Note	This enumeration provides options for controlling the behavior of how authentication affects the ability to launch an Executable.
Aggregated by	Machine.trustedPlatformExecutableLaunchBehavior
Literal	***Description***
monitorMode	An Executable shall always launch, even if the corresponding authentication fails **Tags:** atp.EnumerationLiteralIndex=1
noTrustedPlatform Support	This value shall be used if there is no TrustedPlatform support on the Machine **Tags:** atp.EnumerationLiteralIndex=2
strictMode	An Executable shall not launch if the corresponding authentication fails. **Tags:** atp.EnumerationLiteralIndex=0

Table A.20: TrustedPlatformExecutableLaunchBehaviorEnum

B Platform Extension Interfaces (normative)

This functional cluster does not specify any Platform Extension Interfaces.

C Change history of AUTOSAR traceable items

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

C.1 Traceable item history of this document according to AUTOSAR Release R24-11

C.1.1 Added Specification Items in R24-11

Number	Heading
[SWS_EM_02582]	Single instance of ExecutionClient
[SWS_EM_02583]	Machine State Startup transition result access
[SWS_EM_02584]	SetState access control
[SWS_EM_02585]	Single StateClient instance
[SWS_EM_02586]	Definition of API function ara::exec::FunctionGroup::FunctionGroup

Table C.1: Added Specification Items in R24-11

C.1.2 Changed Specification Items in R24-11

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_01006]	Terminated Process State
[SWS_EM_01012]	Process Argument Passing
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01033]	process start-up configuration
[SWS_EM_01050]	Start Dependent processes
[SWS_EM_01051]	Termination of processes
[SWS_EM_01055]	Initiation of process termination
[SWS_EM_01060]	State transition - termination behavior
[SWS_EM_01065]	State transition - process termination timeout monitoring
[SWS_EM_01066]	State transition - start behavior

▽

△

Number	Heading
[SWS_EM_01078]	Process Argument strings
[SWS_EM_01210]	Report "kRunning received event" to Platform Health Management
[SWS_EM_01211]	Report "initiating process termination" event to Platform Health Management
[SWS_EM_01212]	Report "process terminated" event to Platform Health Management
[SWS_EM_01314]	Default value for terminationBehavior
[SWS_EM_01401]	ExecutionClient usage restriction
[SWS_EM_01404]	Terminating Process State after Termination Request
[SWS_EM_02000]	Definition of API enum ara::exec::ExecutionState
[SWS_EM_02002]	Definition of API function ara::exec::ExecutionClient::~ExecutionClient
[SWS_EM_02003]	Definition of API function ara::exec::ExecutionClient::ReportExecutionState
[SWS_EM_02032]	Behavior on entry to the Unrecoverable State
[SWS_EM_02033]	Behavior after execution of the pre-cleanup action
[SWS_EM_02034]	Behavior after termination of all processes managed by Execution Management
[SWS_EM_02102]	Memory control
[SWS_EM_02103]	CPU usage control
[SWS_EM_02104]	Core affinity
[SWS_EM_02106]	ResourceGroup assignment
[SWS_EM_02108]	Maximum memory usage
[SWS_EM_02109]	process pre-mapping
[SWS_EM_02243]	Handling Execution State Running
[SWS_EM_02245]	Dependency resolution during state change
[SWS_EM_02246]	process specific Environment Variables
[SWS_EM_02247]	Machine specific Environment Variables
[SWS_EM_02250]	Machine State Startup
[SWS_EM_02251]	State transition - restart behavior
[SWS_EM_02253]	State transition - process start-up timeout monitoring
[SWS_EM_02255]	State transition - process termination timeout reaction
[SWS_EM_02258]	State transition - process termination timeout reporting
[SWS_EM_02259]	State transition - process start-up timeout reporting
[SWS_EM_02260]	State transition - process start-up timeout reaction
[SWS_EM_02263]	Definition of API class ara::exec::FunctionGroup
[SWS_EM_02266]	Definition of API function ara::exec::FunctionGroup::~FunctionGroup
[SWS_EM_02267]	Definition of API function ara::exec::FunctionGroup::operator==
[SWS_EM_02268]	Definition of API function ara::exec::FunctionGroup::operator!=
[SWS_EM_02269]	Definition of API class ara::exec::FunctionGroupState
[SWS_EM_02272]	Definition of API function ara::exec::FunctionGroupState::~FunctionGroup State
[SWS_EM_02273]	Definition of API function ara::exec::FunctionGroupState::operator==

▽

△

Number	Heading
[SWS_EM_02274]	Definition of API function ara::exec::FunctionGroupState::operator!=
[SWS_EM_02276]	Definition of API function ara::exec::StateClient::Create
[SWS_EM_02277]	Definition of API function ara::exec::StateClient::~StateClient
[SWS_EM_02278]	Definition of API function ara::exec::StateClient::SetState
[SWS_EM_02279]	Definition of API function ara::exec::StateClient::GetInitialMachineState TransitionResult
[SWS_EM_02281]	Definition of API enum ara::exec::ExecErrc
[SWS_EM_02282]	Definition of API class ara::exec::ExecException
[SWS_EM_02283]	Definition of API function ara::exec::ExecException::ExecException
[SWS_EM_02284]	Definition of API class ara::exec::ExecErrorDomain
[SWS_EM_02286]	Definition of API function ara::exec::ExecErrorDomain::ExecErrorDomain
[SWS_EM_02287]	Definition of API function ara::exec::ExecErrorDomain::Name
[SWS_EM_02288]	Definition of API function ara::exec::ExecErrorDomain::Message
[SWS_EM_02289]	Definition of API function ara::exec::ExecErrorDomain::ThrowAsException
[SWS_EM_02290]	Definition of API function ara::exec::GetExecErrorDomain
[SWS_EM_02291]	Definition of API function ara::exec::MakeErrorCode
[SWS_EM_02296]	Request of a state transition to a state that the Function Group is already in transition to
[SWS_EM_02297]	StateClient usage restriction
[SWS_EM_02298]	Request of a state transition different to the state that the Function Group is already in transition to
[SWS_EM_02301]	Integrity and Authenticity of each `Executable`
[SWS_EM_02302]	Integrity and Authenticity of shared objects
[SWS_EM_02303]	Integrity and Authenticity of processed Execution Manifest configurations
[SWS_EM_02306]	Launch Behavior Validation
[SWS_EM_02307]	Strict Mode - Execution manifest
[SWS_EM_02308]	Strict Mode - Service Instance manifests
[SWS_EM_02309]	Strict Mode - Executables
[SWS_EM_02310]	State transition - process termination after start-up timeout reaction
[SWS_EM_02315]	Unexpected Termination of processes configured for the Requested State during a Function Group State transition
[SWS_EM_02316]	Unexpected Termination of a process not configured for the Requested State during a Function Group State transition
[SWS_EM_02321]	Definition of API function ara::exec::FunctionGroup::FunctionGroup
[SWS_EM_02322]	Definition of API function ara::exec::FunctionGroup::FunctionGroup
[SWS_EM_02324]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02325]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02327]	Definition of API function ara::exec::FunctionGroup::operator=
[SWS_EM_02328]	Definition of API function ara::exec::FunctionGroup::FunctionGroup

▽

△

Number	Heading
[SWS_EM_02329]	Definition of API function ara::exec::FunctionGroup::operator=
[SWS_EM_02330]	Definition of API function ara::exec::FunctionGroupState::operator=
[SWS_EM_02331]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02332]	Definition of API function ara::exec::FunctionGroupState::operator=
[SWS_EM_02400]	Properties of IAM-configuration assigned to processes
[SWS_EM_02541]	Definition of API type ara::exec::ExecutionError
[SWS_EM_02542]	Definition of API function ara::exec::StateClient::GetExecutionError
[SWS_EM_02543]	Default value for ExecutionError
[SWS_EM_02545]	Definition of API variable ara::exec::ExecutionErrorEvent::executionError
[SWS_EM_02546]	Definition of API variable ara::exec::ExecutionErrorEvent::functionGroup
[SWS_EM_02549]	MachineFG.Off handling
[SWS_EM_02552]	State transition - integrity or authenticity check failed
[SWS_EM_02555]	Failure in Machine State Startup transition
[SWS_EM_02557]	Initialization and deinitialization of Execution Management API
[SWS_EM_02558]	Default value for permissionToCreateChildProcess attribute
[SWS_EM_02559]	Restriction of process creation right for processes
[SWS_EM_02560]	Definition of API function ara::exec::ExecutionClient::ExecutionClient
[SWS_EM_02561]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02562]	Definition of API function ara::exec::ExecutionClient::Create
[SWS_EM_02563]	Definition of API function ara::exec::ExecutionClient::ExecutionClient
[SWS_EM_02564]	Definition of API function ara::exec::ExecutionClient::operator=
[SWS_EM_02565]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02566]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02567]	Definition of API function ara::exec::StateClient::operator=
[SWS_EM_02568]	Definition of API function ara::exec::StateClient::operator=
[SWS_EM_02569]	LogMessage ProcessCreated
[SWS_EM_02570]	LogMessage ProcessKRunningReceived
[SWS_EM_02571]	LogMessage ProcessTerminationRequest
[SWS_EM_02572]	LogMessage ProcessTerminated
[SWS_EM_02573]	State Transition logging – process created
[SWS_EM_02574]	State Transition logging – process kRunning received
[SWS_EM_02575]	State Transition logging – process termination request
[SWS_EM_02576]	State Transition logging – process terminated
[SWS_EM_02577]	Call of Termination Handler
[SWS_EM_02578]	Initial signal mask for Reporting Process
[SWS_EM_02579]	Initial signal mask for Non-Reporting process
[SWS_EM_02580]	Definition of API function ara::exec::ExecutionClient::ExecutionClient

▽

△

Number	Heading
[SWS_EM_02581]	Definition of API function ara::exec::ExecutionClient::operator=

Table C.2: Changed Specification Items in R24-11

C.1.3 Deleted Specification Items in R24-11

Number	Heading
[SWS_EM_01013]	Function Group State
[SWS_EM_01030]	Restriction of process creation right for processes
[SWS_EM_01107]	Function Group configuration
[SWS_EM_01301]	Cyclic Execution
[SWS_EM_01302]	Cyclic Execution Control
[SWS_EM_01303]	Cyclic Execution Control Sequence
[SWS_EM_01304]	Service Modification
[SWS_EM_01305]	Worker Pool
[SWS_EM_01306]	processing Container Objects
[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_01320]	Number of `DeterministicClients`
[SWS_EM_01321]	Minimum number of required synchronization requests
[SWS_EM_01322]	Calculation of the next cycle
[SWS_EM_01323]	Total `kRun` loop count
[SWS_EM_01324]	Infinite `kRun` loop
[SWS_EM_01325]	Synchronization Request Message
[SWS_EM_01326]	Synchronization Response Message
[SWS_EM_01327]	Return of the wait point API
[SWS_EM_01328]	Immediate return from wait point
[SWS_EM_01351]	Execution Cycle Time
[SWS_EM_01352]	Execution Cycle Timeout
[SWS_EM_01353]	Event-triggered Cycle Activation
[SWS_EM_02201]	Definition of API enum ara::exec::ActivationReturnType
[SWS_EM_02203]	Definition of API type ara::exec::DeterministicClient::TimeStamp
[SWS_EM_02210]	Definition of API class ara::exec::DeterministicClient
[SWS_EM_02211]	Definition of API function ara::exec::DeterministicClient::DeterministicClient
[SWS_EM_02215]	Definition of API function ara::exec::DeterministicClient::~DeterministicClient

▽

△

Number	Heading
[SWS_EM_02217]	Definition of API function ara::exec::DeterministicClient::WaitForActivation
[SWS_EM_02221]	Definition of API function ara::exec::DeterministicClient::RunWorkerPool
[SWS_EM_02225]	Definition of API function ara::exec::DeterministicClient::GetRandom
[SWS_EM_02226]	Definition of API function ara::exec::DeterministicClient::SetRandomSeed
[SWS_EM_02231]	Definition of API function ara::exec::DeterministicClient::GetActivationTime
[SWS_EM_02236]	Definition of API function ara::exec::DeterministicClient::GetNextActivationTime
[SWS_EM_02254]	Misconfigured process - assigned to more than one Function Group
[SWS_EM_02292]	
[SWS_EM_02313]	`Unexpected Termination` of starting `Processes` during `Function Group State` transition
[SWS_EM_02314]	`Unexpected Termination` of terminating `Processes` during `Function Group State` transition
[SWS_EM_02323]	Definition of API function ara::exec::FunctionGroup::Create
[SWS_EM_02326]	Definition of API function ara::exec::FunctionGroupState::Create
[SWS_EM_02510]	Definition of API class ara::exec::WorkerRunnable
[SWS_EM_02511]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02512]	Definition of API function ara::exec::WorkerRunnable::~WorkerRunnable
[SWS_EM_02513]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02514]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02515]	Definition of API function ara::exec::WorkerRunnable::operator=
[SWS_EM_02520]	Definition of API function ara::exec::WorkerRunnable::Run
[SWS_EM_02530]	Definition of API class ara::exec::WorkerThread
[SWS_EM_02531]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02532]	Definition of API function ara::exec::WorkerThread::~WorkerThread
[SWS_EM_02533]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02534]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02535]	Definition of API function ara::exec::WorkerThread::operator=
[SWS_EM_02540]	Definition of API function ara::exec::WorkerThread::GetRandom
[SWS_EM_02550]	Execution Cycle Termination
[SWS_EM_02551]	Missing DeterministicClient

Table C.3: Deleted Specification Items in R24-11

C.1.4 Added Constraints in R24-11

Number	Heading
[SWS_EM_-CONSTR_-02560]	Function Group shall be controlled by a single State Management process

Table C.4: Added Constraints in R24-11

C.1.5 Changed Constraints in R24-11

Number	Heading
[SWS_EM_-CONSTR_-00001]	Modeling execution dependency for the `Terminated` state
[SWS_EM_-CONSTR_-01744]	Definition of process state in the context of the Execution Dependency
[SWS_EM_-CONSTR_-02556]	Mandatory states
[SWS_EM_-CONSTR_-02557]	Scope of machine Function Group
[SWS_EM_-CONSTR_-02558]	Ability to shut down
[SWS_EM_-CONSTR_-02559]	Ability to restart

Table C.5: Changed Constraints in R24-11

C.1.6 Deleted Constraints in R24-11

none

C.2 Traceable item history of this document according to AUTOSAR Release R23-11

C.2.1 Added Specification Items in R23-11

Number	Heading
[SWS_EM_02295]	Request of a state transition to a state that the `Function Group` is already in
[SWS_EM_02296]	Request of a state transition to a state that the `Function Group` is already in transition to
[SWS_EM_02315]	`Unexpected Termination` of `Processes` configured for the `RequestedState` during a `Function Group State` transition
[SWS_EM_02316]	`Unexpected Termination` of a `Process` not configured for the `RequestedState` during a `Function Group State` transition
[SWS_EM_02511]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02512]	Definition of API function ara::exec::WorkerRunnable::~WorkerRunnable
[SWS_EM_02513]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02514]	Definition of API function ara::exec::WorkerRunnable::WorkerRunnable
[SWS_EM_02515]	Definition of API function ara::exec::WorkerRunnable::operator=
[SWS_EM_02533]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02534]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02535]	Definition of API function ara::exec::WorkerThread::operator=
[SWS_EM_02556]	Monitor Mode
[SWS_EM_02557]	
[SWS_EM_02558]	Default value for permissionToCreateChildProcess attribute
[SWS_EM_02559]	Restriction of process creation right for processes
[SWS_EM_02560]	Definition of API function ara::exec::ExecutionClient::ExecutionClient
[SWS_EM_02561]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02562]	Definition of API function ara::exec::ExecutionClient::Create
[SWS_EM_02563]	Definition of API function ara::exec::ExecutionClient::ExecutionClient
[SWS_EM_02564]	Definition of API function ara::exec::ExecutionClient::operator=
[SWS_EM_02565]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02566]	Definition of API function ara::exec::StateClient::StateClient
[SWS_EM_02567]	Definition of API function ara::exec::StateClient::operator=
[SWS_EM_02568]	Definition of API function ara::exec::StateClient::operator=
[SWS_EM_02569]	LogMessage ProcessCreated
[SWS_EM_02570]	LogMessage ProcessKRunningReceived
[SWS_EM_02571]	LogMessage ProcessTerminationRequest
[SWS_EM_02572]	LogMessage ProcessTerminated
[SWS_EM_02573]	State Transition logging – process created

△

Number	Heading
[SWS_EM_02574]	State Transition logging – process kRunning received
[SWS_EM_02575]	State Transition logging – process termination request
[SWS_EM_02576]	State Transition logging – process terminated
[SWS_EM_02577]	Call of Termination Handler
[SWS_EM_02578]	Initial signal mask for Reporting Process
[SWS_EM_02579]	Initial signal mask for Non-Reporting process
[SWS_EM_02580]	Definition of API function ara::exec::ExecutionClient::ExecutionClient
[SWS_EM_02581]	Definition of API function ara::exec::ExecutionClient::operator=

Table C.6: Added Specification Items in R23-11

C.2.2 Changed Specification Items in R23-11

Number	Heading
[SWS_EM_01030]	Restriction of process creation right for processes
[SWS_EM_01050]	Start Dependent processes
[SWS_EM_01301]	Cyclic Execution
[SWS_EM_01302]	Cyclic Execution Control
[SWS_EM_01303]	Cyclic Execution Control Sequence
[SWS_EM_01304]	Service Modification
[SWS_EM_01305]	Worker Pool
[SWS_EM_01306]	processing Container Objects
[SWS_EM_01309]	`Unexpected Termination` of a `Process`
[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_01320]	Number of `DeterministicClient`s
[SWS_EM_01321]	Minimum number of required synchronization requests
[SWS_EM_01322]	Calculation of the next cycle
[SWS_EM_01323]	Total `kRun` loop count
[SWS_EM_01324]	Infinite `kRun` loop
[SWS_EM_01325]	Synchronization Request Message
[SWS_EM_01326]	Synchronization Response Message
[SWS_EM_01327]	Return of the wait point API
[SWS_EM_01328]	Immediate return from wait point
[SWS_EM_01351]	Execution Cycle Time

▽

△

Number	Heading
[SWS_EM_01352]	Execution Cycle Timeout
[SWS_EM_01353]	Event-triggered Cycle Activation
[SWS_EM_02000]	Definition of API enum ara::exec::ExecutionState
[SWS_EM_02001]	Definition of API class ara::exec::ExecutionClient
[SWS_EM_02002]	Definition of API function ara::exec::ExecutionClient::~ExecutionClient
[SWS_EM_02003]	Definition of API function ara::exec::ExecutionClient::ReportExecutionState
[SWS_EM_02032]	On entry to the `Unrecoverable State`,
[SWS_EM_02033]	After execution of the pre-cleanup action,
[SWS_EM_02034]	After all `Processes` managed by `Execution Management` terminated,
[SWS_EM_02108]	Maximum memory usage
[SWS_EM_02109]	process pre-mapping
[SWS_EM_02201]	Definition of API enum ara::exec::ActivationReturnType
[SWS_EM_02203]	Definition of API type ara::exec::DeterministicClient::TimeStamp
[SWS_EM_02210]	Definition of API class ara::exec::DeterministicClient
[SWS_EM_02211]	Definition of API function ara::exec::DeterministicClient::DeterministicClient
[SWS_EM_02215]	Definition of API function ara::exec::DeterministicClient::~DeterministicClient
[SWS_EM_02217]	Definition of API function ara::exec::DeterministicClient::WaitForActivation
[SWS_EM_02221]	Definition of API function ara::exec::DeterministicClient::RunWorkerPool
[SWS_EM_02225]	Definition of API function ara::exec::DeterministicClient::GetRandom
[SWS_EM_02226]	Definition of API function ara::exec::DeterministicClient::SetRandomSeed
[SWS_EM_02231]	Definition of API function ara::exec::DeterministicClient::GetActivationTime
[SWS_EM_02236]	Definition of API function ara::exec::DeterministicClient::GetNextActivation Time
[SWS_EM_02250]	Machine State Startup
[SWS_EM_02263]	Definition of API class ara::exec::FunctionGroup
[SWS_EM_02266]	Definition of API function ara::exec::FunctionGroup::~FunctionGroup
[SWS_EM_02267]	Definition of API function ara::exec::FunctionGroup::operator==
[SWS_EM_02268]	Definition of API function ara::exec::FunctionGroup::operator!=
[SWS_EM_02269]	Definition of API class ara::exec::FunctionGroupState
[SWS_EM_02272]	Definition of API function ara::exec::FunctionGroupState::~FunctionGroup State
[SWS_EM_02273]	Definition of API function ara::exec::FunctionGroupState::operator==
[SWS_EM_02274]	Definition of API function ara::exec::FunctionGroupState::operator!=
[SWS_EM_02275]	Definition of API class ara::exec::StateClient
[SWS_EM_02276]	Definition of API function ara::exec::StateClient::Create
[SWS_EM_02277]	Definition of API function ara::exec::StateClient::~StateClient
[SWS_EM_02278]	Definition of API function ara::exec::StateClient::SetState
[SWS_EM_02279]	Definition of API function ara::exec::StateClient::GetInitialMachineState TransitionResult
[SWS_EM_02281]	Definition of API enum ara::exec::ExecErrc

▽

△

Number	Heading
[SWS_EM_02282]	Definition of API class ara::exec::ExecException
[SWS_EM_02283]	Definition of API function ara::exec::ExecException::ExecException
[SWS_EM_02284]	Definition of API class ara::exec::ExecErrorDomain
[SWS_EM_02286]	Definition of API function ara::exec::ExecErrorDomain::ExecErrorDomain
[SWS_EM_02287]	Definition of API function ara::exec::ExecErrorDomain::Name
[SWS_EM_02288]	Definition of API function ara::exec::ExecErrorDomain::Message
[SWS_EM_02289]	Definition of API function ara::exec::ExecErrorDomain::ThrowAsException
[SWS_EM_02290]	Definition of API function ara::exec::GetExecErrorDomain
[SWS_EM_02291]	Definition of API function ara::exec::MakeErrorCode
[SWS_EM_02292]	
[SWS_EM_02306]	Launch Behavior Validation
[SWS_EM_02313]	`Unexpected Termination` of starting `Processes` during `Function Group State` transition
[SWS_EM_02314]	`Unexpected Termination` of terminating `Processes` during `Function Group State` transition
[SWS_EM_02321]	Definition of API function ara::exec::FunctionGroup::FunctionGroup
[SWS_EM_02322]	Definition of API function ara::exec::FunctionGroup::FunctionGroup
[SWS_EM_02323]	Definition of API function ara::exec::FunctionGroup::Create
[SWS_EM_02324]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02325]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02326]	Definition of API function ara::exec::FunctionGroupState::Create
[SWS_EM_02327]	Definition of API function ara::exec::FunctionGroup::operator=
[SWS_EM_02328]	Definition of API function ara::exec::FunctionGroup::FunctionGroup
[SWS_EM_02329]	Definition of API function ara::exec::FunctionGroup::operator=
[SWS_EM_02330]	Definition of API function ara::exec::FunctionGroupState::operator=
[SWS_EM_02331]	Definition of API function ara::exec::FunctionGroupState::FunctionGroup State
[SWS_EM_02332]	Definition of API function ara::exec::FunctionGroupState::operator=
[SWS_EM_02510]	Definition of API class ara::exec::WorkerRunnable
[SWS_EM_02520]	Definition of API function ara::exec::WorkerRunnable::Run
[SWS_EM_02530]	Definition of API class ara::exec::WorkerThread
[SWS_EM_02531]	Definition of API function ara::exec::WorkerThread::WorkerThread
[SWS_EM_02532]	Definition of API function ara::exec::WorkerThread::~WorkerThread
[SWS_EM_02540]	Definition of API function ara::exec::WorkerThread::GetRandom
[SWS_EM_02541]	Definition of API type ara::exec::ExecutionError
[SWS_EM_02542]	Definition of API function ara::exec::StateClient::GetExecutionError
[SWS_EM_02544]	Definition of API class ara::exec::ExecutionErrorEvent
[SWS_EM_02545]	Definition of API variable ara::exec::ExecutionErrorEvent::executionError

▽

△

Number	Heading
[SWS_EM_02546]	Definition of API variable ara::exec::ExecutionErrorEvent::functionGroup
[SWS_EM_02547]	Obtain error information
[SWS_EM_02548]	Create error information
[SWS_EM_02550]	Execution Cycle Termination
[SWS_EM_02551]	Missing DeterministicClient

Table C.7: Changed Specification Items in R23-11

C.2.3 Deleted Specification Items in R23-11

Number	Heading
[SWS_EM_02030]	
[SWS_EM_02553]	Rejecting a state transition to a state that the FG is already in
[SWS_EM_02554]	Rejecting a state transition to a state that the FG is already transition to

Table C.8: Deleted Specification Items in R23-11

C.2.4 Added Constraints in R23-11

Number	Heading
[SWS_EM_-CONSTR_-02556]	Mandatory states
[SWS_EM_-CONSTR_-02557]	Scope of machine Function Group
[SWS_EM_-CONSTR_-02558]	Ability to shut down
[SWS_EM_-CONSTR_-02559]	Ability to restart

Table C.9: Added Constraints in R23-11

C.2.5 Changed Constraints in R23-11

none

C.2.6 Deleted Constraints in R23-11

none

C.3 Traceable item history of this document according to AUTOSAR Release R22-11

C.3.1 Added Specification Items in R22-11

Number	Heading
[SWS_EM_01210]	Report “kRunning received event” to Platform Health Management
[SWS_EM_01211]	Report “initiating process termination” event to Platform Health Management
[SWS_EM_01212]	Report “process terminated” event to Platform Health Management
[SWS_EM_02552]	State transition - integrity or authenticity check failed
[SWS_EM_02553]	Rejecting a state transition to a state that the FG is already in
[SWS_EM_02554]	Rejecting a state transition to a state that the FG is already transition to
[SWS_EM_02555]	Failure in Machine State Startup transition
[SWS_EM_-CONSTR_00001]	Modeling execution dependency for the `Terminated` state
[SWS_EM_-CONSTR_01744]	Definition of process state in the context of the `ExecutionDependency`

Table C.10: Added Specification Items in R22-11

C.3.2 Changed Specification Items in R22-11

Number	Heading
[SWS_EM_01013]	Function Group State
[SWS_EM_01067]	Actions on Completion State Transition
[SWS_EM_01078]	Process Argument strings
[SWS_EM_01107]	Function Group configuration
[SWS_EM_01110]	Off States
[SWS_EM_01314]	Default value for `terminationBehavior`
[SWS_EM_01320]	Number of `DeterministicClient`s
[SWS_EM_01321]	Minimum number of required synchronization requests
[SWS_EM_01322]	Calculation of the next cycle
[SWS_EM_01323]	Total `kRun` loop count
[SWS_EM_01324]	Infinite `kRun` loop

▽

△

Number	Heading
[SWS_EM_01403]	Reporting Non-reporting Process
[SWS_EM_02000]	
[SWS_EM_02001]	
[SWS_EM_02002]	
[SWS_EM_02003]	
[SWS_EM_02030]	
[SWS_EM_02108]	Maximum memory usage
[SWS_EM_02201]	
[SWS_EM_02203]	
[SWS_EM_02210]	
[SWS_EM_02211]	
[SWS_EM_02215]	
[SWS_EM_02217]	
[SWS_EM_02221]	
[SWS_EM_02225]	
[SWS_EM_02226]	
[SWS_EM_02231]	
[SWS_EM_02236]	
[SWS_EM_02241]	Machine State Startup Completion
[SWS_EM_02254]	Misconfigured process - assigned to more than one Function Group
[SWS_EM_02263]	
[SWS_EM_02266]	
[SWS_EM_02267]	
[SWS_EM_02268]	
[SWS_EM_02269]	
[SWS_EM_02272]	
[SWS_EM_02273]	
[SWS_EM_02274]	
[SWS_EM_02275]	
[SWS_EM_02276]	
[SWS_EM_02277]	
[SWS_EM_02278]	
[SWS_EM_02279]	
[SWS_EM_02280]	Effect on Execution Dependency
[SWS_EM_02281]	
[SWS_EM_02282]	
[SWS_EM_02283]	
[SWS_EM_02284]	
[SWS_EM_02286]	

▽

△

Number	Heading
[SWS_EM_02287]	
[SWS_EM_02288]	
[SWS_EM_02289]	
[SWS_EM_02290]	
[SWS_EM_02291]	
[SWS_EM_02292]	
[SWS_EM_02297]	StateClient usage restriction
[SWS_EM_02298]	Canceling ongoing state transition
[SWS_EM_02299]	Availability of a Trust Anchor
[SWS_EM_02306]	Launch Behavior Validation
[SWS_EM_02321]	
[SWS_EM_02322]	
[SWS_EM_02323]	
[SWS_EM_02324]	
[SWS_EM_02325]	
[SWS_EM_02326]	
[SWS_EM_02327]	
[SWS_EM_02328]	
[SWS_EM_02329]	
[SWS_EM_02330]	
[SWS_EM_02331]	
[SWS_EM_02332]	
[SWS_EM_02400]	Properties of IAM-configuration assigned to processes
[SWS_EM_02510]	
[SWS_EM_02520]	
[SWS_EM_02530]	
[SWS_EM_02531]	
[SWS_EM_02532]	
[SWS_EM_02540]	
[SWS_EM_02541]	
[SWS_EM_02542]	
[SWS_EM_02544]	
[SWS_EM_02545]	
[SWS_EM_02546]	

Table C.11: Changed Specification Items in R22-11

C.3.3 Deleted Specification Items in R22-11

Number	Heading
[SWS_EM_01308]	Random Numbers
[SWS_EM_02107]	Maximum heap
[SWS_EM_02304]	Integrity and Authenticity of processed `Service Instance Manifests`
[SWS_EM_-CONSTR_0001]	Modeling execution dependency for the `Terminated` state
[SWS_EM_-CONSTR_1744]	Definition of process state in the context of the `ExecutionDependency`

Table C.12: Deleted Specification Items in R22-11

C.3.4 Added Constraints in R22-11

none

C.3.5 Changed Constraints in R22-11

none

C.3.6 Deleted Constraints in R22-11

none

C.4 Traceable item history of this document according to AUTOSAR Release R21-11

C.4.1 Added Specification Items in R21-11

Number	Heading
[SWS_EM_02547]	Obtain error information
[SWS_EM_02548]	Create error information
[SWS_EM_CON-STR_1744]	Definition of process state in the context of the ExecutionDependency
[SWS_EM_CON-STR_0001]	Modeling execution dependency for the Terminated state

▽

△

Number	Heading
[SWS_EM_02549]	MachineFG.Off handling
[SWS_EM_02551]	Missing DeterministicClient
[SWS_EM_02550]	Execution Cycle Termination
[SWS_EM_01328]	Immediate return from wait point
[SWS_EM_02323]	FunctionGroup::Create
[SWS_EM_02321]	FunctionGroup::FunctionGroup
[SWS_EM_02322]	FunctionGroup::FunctionGroup (Copy Constructor)
[SWS_EM_02328]	FunctionGroup::FunctionGroup (Move Constructor)
[SWS_EM_02327]	FunctionGroup::operator= (Copy assignment operator)
[SWS_EM_02329]	FunctionGroup::operator= (Move assignment operator)
[SWS_EM_02326]	FunctionGroupState::Create
[SWS_EM_02324]	FunctionGroupState::FunctionGroupState
[SWS_EM_02325]	FunctionGroupState::FunctionGroupState (Copy Constructor)
[SWS_EM_02331]	FunctionGroupState::FunctionGroupState (Move Constructor)
[SWS_EM_02330]	FunctionGroupState::operator= (Copy assignment operator)
[SWS_EM_02332]	FunctionGroupState::operator= (Move assignment operator)

Table C.13: Added Specification Items in R21-11

C.4.2 Changed Specification Items in R21-11

Number	Heading
[SWS_EM_01309]	Unexpected Termination of a process
[SWS_EM_02243]	Handling Execution State Running
[SWS_EM_01032]	Machine States configuration
[SWS_EM_02241]	Machine State Startup Completion
[SWS_EM_02254]	Misconfigured process - assigned to more than one Function Group
[SWS_EM_01060]	State transition - termination behavior
[SWS_EM_02255]	State transition - process termination timeout reaction
[SWS_EM_02258]	State transition - process termination timeout reporting
[SWS_EM_02313]	Unexpected Termination of starting processes during Function Group State transition
[SWS_EM_02314]	Unexpected Termination of terminating processes during Function Group State transition
[SWS_EM_01351]	Execution Cycle Time
[SWS_EM_01352]	Execution Cycle Timeout
[SWS_EM_01353]	Event-triggered Cycle Activation

▽

△

Number	Heading
[SWS_EM_01308]	Random Numbers
[SWS_EM_01313]	Next Activation Time Unknown
[SWS_EM_02102]	Memory control
[SWS_EM_02103]	CPU usage control
[SWS_EM_02104]	Core affinity
[SWS_EM_01014]	Scheduling policy
[SWS_EM_02107]	Maximum heap
[SWS_EM_02108]	Maximum system memory usage
[SWS_EM_02109]	process pre-mapping
[SWS_EM_02300]	Integrity and Authenticity of Machine configuration
[SWS_EM_02303]	Integrity and Authenticity of processed Execution Manifest configurations
[SWS_EM_02304]	Integrity and Authenticity of processed Service Instance Manifests
[SWS_EM_02305]	Failed authenticity checks
[SWS_EM_02306]	Launch Behavior Validation
[SWS_EM_02309]	Strict Mode - Executables
[SWS_EM_02276]	StateClient::StateClient
[SWS_EM_02281]	Execution Management error codes

Table C.14: Changed Specification Items in R21-11

C.4.3 Deleted Specification Items in R21-11

Number	Heading
[SWS_EM_01405]	Terminating Process State after Terminating Report
[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_01109]	Misconfigured Process - not assigned to a Function Group
[SWS_EM_01307]	Worker Object
[SWS_EM_02202]	ActivationTimeStampReturnType
[SWS_EM_02216]	DeterministicClient::WaitForNextActivation
[SWS_EM_02220]	DeterministicClient::RunWorkerPool
[SWS_EM_02230]	DeterministicClient::GetActivationTime
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime
[SWS_EM_02264]	FunctionGroup::Preconstruct

▽

△

Number	Heading
[SWS_EM_02265]	FunctionGroup::FunctionGroup
[SWS_EM_02270]	FunctionGroupState::Preconstruct
[SWS_EM_02271]	FunctionGroupState::FunctionGroupState

Table C.15: Deleted Specification Items in R21-11

C.4.4 Added Constraints in R21-11

none

C.4.5 Changed Constraints in R21-11

none

C.4.6 Deleted Constraints in R21-11

none

C.5 Traceable item history of this document according to AUTOSAR Release R20-11

C.5.1 Added Specification Items in R20-11

Number	Heading
[SWS_EM_01314]	Default value for terminationBehavior
[SWS_EM_01309]	Unexpected Termination of a process
[SWS_EM_01078]	Process Argument strings
[SWS_EM_02311]	Order of process termination timeout reaction
[SWS_EM_02310]	State transition - process termination after start-up timeout reaction
[SWS_EM_02312]	Order of process start-up timeout reaction
[SWS_EM_02280]	Effect on Execution Dependency
[SWS_EM_02313]	Unexpected Termination of starting processes during Function Group State transition
[SWS_EM_02314]	Unexpected Termination of terminating processes during Function Group State transition

▽

△

Number	Heading
[SWS_EM_01320]	Number of DeterministicClients
[SWS_EM_01321]	Minimum number of required synchronization requests
[SWS_EM_01322]	Calculation of the next cycle
[SWS_EM_01323]	Total kRun loop count
[SWS_EM_01324]	Infinite kRun loop
[SWS_EM_01325]	Synchronization Request Message
[SWS_EM_01326]	Synchronization Response Message
[SWS_EM_01327]	Return of the wait point API
[SWS_EM_02032]	On entry to the Unrecoverable State
[SWS_EM_02033]	After execution of the pre-cleanup action
[SWS_EM_02034]	After all processes managed by Execution Management terminated,
[SWS_EM_02400]	Properties of IAM-configuration assigned to processes
[SWS_EM_02203]	DeterministicClient::TimeStamp
[SWS_EM_02541]	ExecutionError
[SWS_EM_02544]	ExecutionErrorEvent
[SWS_EM_02545]	ExecutionErrorEvent::executionError
[SWS_EM_02546]	ExecutionErrorEvent::functionGroup
[SWS_EM_02510]	WorkerRunnable class
[SWS_EM_02520]	WorkerRunnable::Run
[SWS_EM_02530]	WorkerThread class
[SWS_EM_02531]	WorkerThread::WorkerThread
[SWS_EM_02532]	WorkerThread:: WorkerThread
[SWS_EM_02540]	WorkerThread::GetRandom
[SWS_EM_02217]	DeterministicClient::WaitForActivation
[SWS_EM_02221]	DeterministicClient::RunWorkerPool
[SWS_EM_02226]	DeterministicClient::SetRandomSeed
[SWS_EM_02231]	DeterministicClient::GetActivationTime
[SWS_EM_02236]	DeterministicClient::GetNextActivationTime
[SWS_EM_02542]	StateClient::GetExecutionError
[SWS_EM_02543]	Default value for ExecutionError
[SWS_EM_02290]	GetExecErrorDomain
[SWS_EM_02291]	MakeErrorCode

Table C.16: Added Specification Items in R20-11

C.5.2 Changed Specification Items in R20-11

Number	Heading
[SWS_EM_02243]	Handling Execution State Running
[SWS_EM_01405]	Terminating Process State after Terminating Report
[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_01032]	Machine States configuration
[SWS_EM_02250]	Machine State Startup
[SWS_EM_02258]	State transition - Process termination timeout reporting
[SWS_EM_02260]	State transition - Process start-up timeout reaction
[SWS_EM_02259]	State transition - Process start-up timeout reporting
[SWS_EM_01067]	Confirm State Changes
[SWS_EM_02297]	StateClient usage restriction
[SWS_EM_02000]	ExecutionState
[SWS_EM_02202]	ActivationTimeStampReturnType
[SWS_EM_02216]	DeterministicClient::WaitForNextActivation
[SWS_EM_02220]	DeterministicClient::RunWorkerPool
[SWS_EM_02230]	DeterministicClient::GetActivationTime
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime
[SWS_EM_02263]	FunctionGroup class
[SWS_EM_02264]	FunctionGroup::Preconstruct
[SWS_EM_02265]	FunctionGroup::FunctionGroup
[SWS_EM_02266]	FunctionGroup:: FunctionGroup
[SWS_EM_02267]	FunctionGroup::operator==
[SWS_EM_02268]	FunctionGroup::operator!=
[SWS_EM_02269]	FunctionGroupState class
[SWS_EM_02270]	FunctionGroupState::Preconstruct
[SWS_EM_02271]	FunctionGroupState::FunctionGroupState
[SWS_EM_02272]	FunctionGroupState:: FunctionGroupState
[SWS_EM_02273]	FunctionGroupState::operator==
[SWS_EM_02274]	FunctionGroupState::operator!=

Table C.17: Changed Specification Items in R20-11

C.5.3 Deleted Specification Items in R20-11

Number	Heading
[SWS_EM_01071]	Premature Termination of a Reporting Process
[SWS_EM_02244]	Handling Execution State Terminating
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_02242]	Further Function Group State Changes
[SWS_EM_02257]	Recovery Action API Security
[SWS_EM_02076]	Get Process States Information
[SWS_EM_02077]	Process State Transition Event
[SWS_EM_01016]	Process Restart
[SWS_EM_01062]	Process Restart Behavior
[SWS_EM_01063]	Process Restart Failed
[SWS_EM_01064]	Process Restart Successful
[SWS_EM_02261]	Enter Unrecoverable State
[SWS_EM_02262]	Enter Unrecoverable State Behavior

Table C.18: Deleted Specification Items in R20-11

C.5.4 Added Constraints in R20-11

none

C.5.5 Changed Constraints in R20-11

none

C.5.6 Deleted Constraints in R20-11

none

C.6 Traceable item history of this document according to AUTOSAR Release R19-11

C.6.1 Added Specification Items in R19-11

Number	Heading
[SWS_EM_01401]	Process Self Reporting
[SWS_EM_01402]	Implicit Running Process State
[SWS_EM_01403]	Reporting Non-reporting Process
[SWS_EM_01404]	Terminating Process State after Termination Request
[SWS_EM_01405]	Terminating Process State after Terminating Report
[SWS_EM_02002]	
[SWS_EM_02003]	
[SWS_EM_02030]	
[SWS_EM_02211]	
[SWS_EM_02215]	
[SWS_EM_02216]	
[SWS_EM_02220]	
[SWS_EM_02225]	
[SWS_EM_02230]	
[SWS_EM_02235]	
[SWS_EM_02257]	Recovery Action API Security
[SWS_EM_02258]	State transition - Process termination timeout reporting
[SWS_EM_02259]	State transition - Process start-up timeout reporting
[SWS_EM_02260]	State transition - Process start-up timeout reaction
[SWS_EM_02261]	Enter Unrecoverable State
[SWS_EM_02262]	Enter Unrecoverable State Behavior
[SWS_EM_02263]	
[SWS_EM_02264]	
[SWS_EM_02265]	
[SWS_EM_02266]	
[SWS_EM_02267]	
[SWS_EM_02268]	
[SWS_EM_02269]	
[SWS_EM_02270]	
[SWS_EM_02271]	
[SWS_EM_02272]	
[SWS_EM_02273]	
[SWS_EM_02274]	

▽

△

Number	Heading
[SWS_EM_02275]	
[SWS_EM_02276]	
[SWS_EM_02277]	
[SWS_EM_02278]	
[SWS_EM_02279]	
[SWS_EM_02281]	
[SWS_EM_02282]	
[SWS_EM_02283]	
[SWS_EM_02284]	
[SWS_EM_02286]	
[SWS_EM_02287]	
[SWS_EM_02288]	
[SWS_EM_02289]	
[SWS_EM_02290]	
[SWS_EM_02291]	
[SWS_EM_02292]	
[SWS_EM_02297]	StateClient usage restriction
[SWS_EM_02298]	Canceling ongoing state transition
[SWS_EM_02299]	Availability of a Trust Anchor
[SWS_EM_02300]	Integrity and Authenticity of processed `Machine Manifest`
[SWS_EM_02301]	Integrity and Authenticity of each `Executable`
[SWS_EM_02302]	Integrity and Authenticity of shared objects
[SWS_EM_02303]	Integrity and Authenticity of processed `Execution Manifests`
[SWS_EM_02304]	Integrity and Authenticity of processed `Service Instance Manifests`
[SWS_EM_02305]	Failed authenticity checks
[SWS_EM_02306]	Machine Manifest
[SWS_EM_02307]	Strict Mode - Execution manifest
[SWS_EM_02308]	Strict Mode - Service Instance manifests
[SWS_EM_02309]	Strict Mode - Executables

Table C.19: Added Specification Items in R19-11

C.6.2 Changed Specification Items in R19-11

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 of Reporting Processes
[SWS_EM_01006]	Terminated Process State
[SWS_EM_01012]	Process Argument Passing
[SWS_EM_01013]	Function Group State
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01016]	Process Restart
[SWS_EM_01023]	Self initiation of Machine State Startup transition
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_01025]	Machine State Restart
[SWS_EM_01030]	Restriction of process creation right for Processes
[SWS_EM_01032]	Machine States configuration
[SWS_EM_01033]	Process start-up configuration
[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]	Termination of Processes
[SWS_EM_01055]	Initiation of Process termination
[SWS_EM_01060]	State transition - termination behavior
[SWS_EM_01062]	Process Restart Behavior
[SWS_EM_01063]	Process Restart Failed
[SWS_EM_01064]	Process Restart Successful
[SWS_EM_01065]	State transition - Process termination timeout monitoring
[SWS_EM_01066]	State transition - start behavior
[SWS_EM_01067]	Finish of a successful state transition
[SWS_EM_01071]	Premature Termination of a Reporting Process
[SWS_EM_01072]	Process 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

▽

△

Number	Heading
[SWS_EM_01077]	Long form Arguments without option value
[SWS_EM_01107]	Function Group configuration
[SWS_EM_01109]	Misconfigured Process - not assigned to a Function Group
[SWS_EM_01110]	Off States
[SWS_EM_01301]	Cyclic Execution
[SWS_EM_01302]	Cyclic Execution Control
[SWS_EM_01303]	Cyclic Execution Control Sequence
[SWS_EM_01304]	Service Modification
[SWS_EM_01305]	Worker Pool
[SWS_EM_01306]	Processing Container Objects
[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_01351]	Execution Cycle Time
[SWS_EM_01352]	Execution Cycle Timeout
[SWS_EM_01353]	Event-triggered Cycle Activation
[SWS_EM_02076]	Get Process States Information
[SWS_EM_02077]	Process State Transition Event
[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_02241]	Machine State Startup Completion
[SWS_EM_02242]	Further Function Group State Changes
[SWS_EM_02243]	Handling Execution State Running
[SWS_EM_02244]	Handling Execution State Terminating
[SWS_EM_02245]	Dependency resolution during state change
[SWS_EM_02246]	Process specific Environment Variables
[SWS_EM_02247]	Machine specific Environment Variables
[SWS_EM_02248]	Environment Variables precedence
[SWS_EM_02249]	Missing value from Environment Variable definition
[SWS_EM_02250]	Machine State Startup
[SWS_EM_02251]	State transition - restart behavior
[SWS_EM_02253]	State transition - Process start-up timeout monitoring

▽

△

Number	Heading
[SWS_EM_02254]	Misconfigured Process - assigned to more than one Function Group
[SWS_EM_02255]	State transition - Process termination timeout reaction

Table C.20: Changed Specification Items in R19-11

C.6.3 Deleted Specification Items in R19-11

Number	Heading
[SWS_EM_01005]	Terminating Process State
[SWS_EM_01018]	Enter Safe State
[SWS_EM_01026]	State Change
[SWS_EM_01028]	Get State Information
[SWS_EM_01034]	Deny State Change Request
[SWS_EM_01053]	Execution State Running
[SWS_EM_01061]	Enter Safe State Behavior
[SWS_EM_01068]	State transition - Process start-up timeout reporting
[SWS_EM_01070]	Acknowledgement of termination request
[SWS_EM_01400]	Execution Dependency resolution
[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_02058]	State Transition Timeout
[SWS_EM_02252]	State transition - Process termination timeout reporting
[SWS_EM_02256]	State transition - Process start-up timeout reaction

Table C.21: Deleted Specification Items in R19-11

C.6.4 Added Constraints in R19-11

none

C.6.5 Changed Constraints in R19-11

none

C.6.6 Deleted Constraints in R19-11

none

C.7 Traceable item history of this document according to AUTOSAR Release 19-03

C.7.1 Added Specification Items in R19-03

Number	Heading
[SWS_EM_02250]	Machine State Startup
[SWS_EM_02251]	State transition - restart behavior
[SWS_EM_02252]	State transition - Process termination timeout reporting
[SWS_EM_02253]	State transition - Process start-up timeout monitoring
[SWS_EM_02254]	Misconfigured Process - assigned to more than one Function Group
[SWS_EM_02255]	State transition - Process termination timeout reaction
[SWS_EM_02256]	State transition - Process start-up timeout reaction

Table C.22: Added Specification Items in R19-03

C.7.2 Changed Specification Items in R19-03

Number	Heading
[SWS_EM_01001]	Execution Dependency error
[SWS_EM_01005]	Terminating Process State
[SWS_EM_01012]	Process Argument Passing
[SWS_EM_01013]	Function Group State
[SWS_EM_01014]	Scheduling policy
[SWS_EM_01015]	Scheduling priority
[SWS_EM_01023]	Self initiation of Machine State Startup transition
[SWS_EM_01024]	Machine State Shutdown
[SWS_EM_01025]	Machine State Restart
[SWS_EM_01060]	State transition - termination behavior
[SWS_EM_01065]	State transition - Process termination timeout monitoring
[SWS_EM_01066]	State transition - start behavior
[SWS_EM_01067]	Finish of a successful state transition
[SWS_EM_01068]	State transition - Process start-up timeout reporting
[SWS_EM_01109]	Misconfigured Process - not assigned to a Function Group

△

Number	Heading
[SWS_EM_01110]	Off States
[SWS_EM_01400]	Execution Dependency resolution
[SWS_EM_02000]	
[SWS_EM_02001]	
[SWS_EM_02201]	
[SWS_EM_02202]	
[SWS_EM_02210]	
[SWS_EM_02241]	Machine State Startup Completion
[SWS_EM_02245]	Dependency resolution during state change
[SWS_EM_02246]	Process specific Environment Variables

Table C.23: Changed Specification Items in R19-03

C.7.3 Deleted Specification Items in R19-03

Number	Heading
[SWS_EM_01035]	Machine State Restart behavior
[SWS_EM_01036]	Machine State Shutdown behavior
[SWS_EM_02002]	ExecutionClient::~ExecutionClient API
[SWS_EM_02003]	ExecutionClient::ReportExecutionState API
[SWS_EM_02030]	ExecutionClient::ExecutionClient API
[SWS_EM_02070]	ExecutionReturnType Enumeration
[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
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime API

Table C.24: Deleted Specification Items in R19-03

C.7.4 Added Constraints in R19-03

none

C.7.5 Changed Constraints in R19-03

none

C.7.6 Deleted Constraints in R19-03

none

C.8 Traceable item history of this document according to AUTOSAR Release 18-10

C.8.1 Added Specification Items in 18-10

none

C.8.2 Changed Specification Items 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

▽

△

Number	Heading
[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
[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.25: Changed Specification Items in 18-10

C.8.3 Deleted Specification Items 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.26: Deleted Specification Items in 18-10

C.8.4 Added Constraints in 18-10

none

C.8.5 Changed Constraints in 18-10

none

C.8.6 Deleted Constraints in 18-10

none

C.9 Traceable item history of this document according to AUTOSAR Release 18-03

C.9.1 Added Specification Items 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
[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

▽

△

Number	Heading
[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
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime API

Table C.27: Added Specification Items in 18-03

C.9.2 Changed Specification Items 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
[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

△

Number	Heading
[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.28: Changed Specification Items in 18-03

C.9.3 Deleted Specification Items 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
[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

▽

△

Number	Heading
[SWS_EM_02073]	Retrieving Function Group State
[SWS_EM_02074]	Setting Machine State
[SWS_EM_02075]	Setting Function Group State

Table C.29: Deleted Specification Items in 18-03

C.9.4 Added Constraints in 18-03

none

C.9.5 Changed Constraints in 18-03

none

C.9.6 Deleted Constraints in 18-03

none

C.10 Traceable item history of this document according to AUTOSAR Release 17-10

C.10.1 Added Specification Items 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

▽

△

Number	Heading
[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
[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.30: Added Specification Items in 17-10

C.10.2 Changed Specification Items 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

▽

△

Number	Heading
[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
[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.31: Changed Specification Items in 17-10

C.10.3 Deleted Specification Items 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.32: Deleted Specification Items in 17-10

C.10.4 Added Constraints in 17-10

none

C.10.5 Changed Constraints in 17-10

none

C.10.6 Deleted Constraints in 17-10

none

C.11 Traceable item history of this document according to AUTOSAR Release 17-03

C.11.1 Added Specification Items in 17-03

Number	Heading
[SWS_EM_00017]	Application Processes
[SWS_EM_01000]	Startup order
[SWS_EM_01002]	Idle Process State

▽

△

Number	Heading
[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
[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`
[SWS_EM_01026]	`Machine State` change
[SWS_EM_01027]	Rejection of Client Requests
[SWS_EM_01028]	`GetMachineState` API
[SWS_EM_01029]	`SetMachineState` API
[SWS_EM_01030]	Start of Application execution
[SWS_EM_01033]	Application start-up configuration
[SWS_EM_01034]	Deny `SetMachineState` API 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 Executable`s
[SWS_EM_01051]	Shutdown `Application Executable`s
[SWS_EM_01052]	Application State `Initializing`
[SWS_EM_01053]	Application State `Running`
[SWS_EM_01055]	Application State `Termination`
[SWS_EM_01056]	Machine State Management Application
[SWS_EM_01057]	Machine State Change arbitration
[SWS_EM_01058]	Shutdown of the Operating System
[SWS_EM_01059]	Restart of the Operating System
[SWS_EM_01060]	`Machine State` change behavior
[SWS_EM_02000]	

▽

△

Number	Heading
[SWS_EM_02001]	
[SWS_EM_02002]	
[SWS_EM_02003]	
[SWS_EM_02005]	
[SWS_EM_02006]	
[SWS_EM_02007]	
[SWS_EM_02008]	
[SWS_EM_02009]	
[SWS_EM_02014]	
[SWS_EM_02019]	
[SWS_EM_02030]	
[SWS_EM_02031]	Application State Reporting
[SWS_EM_99999]	
[SWS_OSI_01007]	

Table C.33: Added Specification Items in 17-03

C.11.2 Changed Specification Items in 17-03

none

C.11.3 Deleted Specification Items in 17-03

none

C.11.4 Added Constraints in 17-03

none

C.11.5 Changed Constraints in 17-03

none

C.11.6 Deleted Constraints in 17-03

none