

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	R19-11

Document Change History			
Date	Release	Changed by	Description
2019-11-28	R19-11	AUTOSAR Release	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Refinement of State Management semantics • Document structure modified to reflect current template
2018-10-31	18-10	AUTOSAR Release Management	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Deterministic Execution • Resource Limitation • State Management • Fault Tolerance elaboration

2017-10-27	17-10	AUTOSAR Release Management	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Requirement Levels

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as follows, based on [1].

Note that the requirement level of the document in which they are used modifies the force of these words.

- **MUST:** This word, or the adjective "LEGALLY REQUIRED", means that the definition is an absolute requirement of the specification due to legal issues.
- **MUST NOT:** This phrase, or the phrase "MUST NOT", means that the definition is an absolute prohibition of the specification due to legal issues.
- **SHALL:** This phrase, or the adjective "REQUIRED", means that the definition is an absolute requirement of the specification.
- **SHALL NOT:** This phrase means that the definition is an absolute prohibition of the specification.
- **SHOULD:** This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **SHOULD NOT:** This phrase, or the phrase "NOT RECOMMENDED", means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood

and the case carefully weighed before implementing any behavior described with this label.

- **MAY:** This word, or the adjective "OPTIONAL", means that an item is truly optional. One vendor may choose to include the item because a particular market-place requires it or because the vendor feels that it enhances the product while another vendor may omit the same item.

An implementation, which does not include a particular option, **SHALL** be prepared to interoperate with another implementation, which does include the option, though perhaps with reduced functionality. In the same vein an implementation, which does include a particular option, **SHALL** be prepared to interoperate with another implementation, which does not include the option (except, of course, for the feature the option provides.)

Table of Contents

1	Introduction and functional overview	10
1.1	What is Execution Management?	10
1.2	Interaction with AUTOSAR Runtime for Adaptive	10
2	Acronyms and abbreviations	12
3	Related documentation	15
3.1	Input documents & related standards and norms	15
3.2	Further applicable specification	16
4	Constraints and assumptions	17
4.1	Known Limitations	17
5	Dependencies to other Functional Clusters	18
5.1	Protocol layer dependencies	18
6	Requirements traceability	19
6.1	Not applicable requirements	21
7	Functional specification	22
7.1	Functional Cluster Lifecycle	23
7.1.1	Startup	23
7.1.2	Shutdown	23
7.1.3	Restart	23
7.2	Technical Overview	23
7.2.1	Application	23
7.2.2	Adaptive Application	23
7.2.3	Executable	24
7.2.4	Process	25
7.2.5	Execution Manifest	26
7.2.6	Machine Manifest	26
7.2.7	Manifest Format	27
7.3	Execution Management Responsibilities	28
7.4	Process Lifecycle Management	29
7.4.1	Execution State	29
7.4.2	Process States	31
7.4.3	Startup and Termination	32
7.4.3.1	Ordering	32
7.4.3.2	Arguments	33
7.4.3.3	Environment Variables	35
7.4.4	Startup Sequence	35
7.4.4.1	Execution Dependency	36
7.5	State Management	40
7.5.1	Overview	40
7.5.2	Machine State	40

7.5.2.1	Startup	42
7.5.2.2	Shutdown	43
7.5.2.3	Restart	44
7.5.3	Function Group State	44
7.5.4	State Interaction	47
7.5.5	State Transition	48
7.6	Application Recovery Actions	54
7.6.1	Overview	54
7.6.2	Process State Information	55
7.6.2.1	Get Process States Information	55
7.6.2.2	Process State Transition Event	55
7.6.3	Recovery Actions	55
7.6.3.1	Process Restart	55
7.6.3.2	Enter Unrecoverable State	55
7.7	Deterministic Execution	57
7.7.1	Determinism	57
7.7.1.1	Time Determinism	58
7.7.1.2	Data Determinism	58
7.7.1.3	Full Determinism	58
7.7.2	Redundant Deterministic Execution	59
7.7.3	Cyclic Deterministic Execution	62
7.7.3.1	Control of Cyclic Execution	63
7.7.3.2	Worker Pool	65
7.7.3.3	Random Numbers	67
7.7.3.4	Time Stamps	67
7.7.3.5	Real-Time Resources	68
7.8	Resource Limitation	72
7.8.1	Resource Configuration	72
7.8.2	Resource Monitoring	74
7.8.3	Application-level Resource configuration	75
7.8.3.1	CPU Usage	75
7.8.3.2	Core Affinity	75
7.8.3.3	Scheduling	76
7.8.3.4	Memory Budget and Monitoring	77
7.8.3.5	Working Folder	78
7.9	Fault Tolerance	79
7.9.1	Introduction	79
7.9.2	Scope	79
7.9.3	Threat Model	79
7.10	Security	81
7.10.1	Trusted Platform	81
7.10.1.1	Handling of failed authenticity checks	83
8	API specification	85
8.1	Type Definitions	85
8.1.1	ExecutionState	85

8.1.2	ActivationReturnType	85
8.1.3	ActivationTimeStampReturnType	86
8.2	Class Definitions	86
8.2.1	ExecutionClient class	86
8.2.1.1	ExecutionClient::ExecutionClient	87
8.2.1.2	ExecutionClient::~~ExecutionClient	87
8.2.1.3	ExecutionClient::ReportExecutionState	87
8.2.2	DeterministicClient class	88
8.2.2.1	DeterministicClient::DeterministicClient	88
8.2.2.2	DeterministicClient::~~DeterministicClient	89
8.2.2.3	DeterministicClient::WaitForNextActivation	89
8.2.2.4	DeterministicClient::RunWorkerPool	89
8.2.2.5	DeterministicClient::GetRandom	90
8.2.2.6	DeterministicClient::GetActivationTime	90
8.2.2.7	DeterministicClient::GetNextActivationTime	91
8.2.3	FunctionGroup class	91
8.2.3.1	FunctionGroup::Preconstruct	92
8.2.3.2	FunctionGroup::FunctionGroup	92
8.2.3.3	FunctionGroup::~~FunctionGroup	93
8.2.3.4	FunctionGroup::operator==	93
8.2.3.5	FunctionGroup::operator!=	94
8.2.4	FunctionGroupState class	94
8.2.4.1	FunctionGroupState::Preconstruct	95
8.2.4.2	FunctionGroupState::FunctionGroupState	95
8.2.4.3	FunctionGroupState::~~FunctionGroupState	96
8.2.4.4	FunctionGroupState::operator==	96
8.2.4.5	FunctionGroupState::operator!=	96
8.2.5	StateClient class	97
8.2.5.1	StateClient::StateClient	97
8.2.5.2	StateClient::~~StateClient	98
8.2.5.3	StateClient::SetState	98
8.2.5.4	StateClient::GetInitialMachineStateTransitionResult	99
8.3	Errors	101
8.3.1	Execution Management error codes	101
8.3.2	ExecException type	101
8.3.2.1	ExecException::ExecException	102
8.3.3	GetExecErrorDomain function	102
8.3.4	MakeErrorCode function	102
8.3.5	ExecErrorDomain type	103
8.3.5.1	ExecErrorDomain::ExecErrorDomain	103
8.3.5.2	ExecErrorDomain::Name	104
8.3.5.3	ExecErrorDomain::Message	104
8.3.5.4	ExecErrorDomain::ThrowAsException	104
9	Service Interfaces	106
A	Mentioned Manifest Elements	107

B	Interfaces to other Functional Clusters (informative)	114
B.1	Overview	114
B.2	Interface Tables	115
B.2.1	Get Process States Information	115
B.2.2	Enter Unrecoverable State Request	116
B.2.3	Process Restart Request	116
C	History of Constraints and Specification Items	117
C.1	Constraint and Specification Item History of this document according to AUTOSAR Release 17-10	117
C.1.1	Added Traceables in 17-10	117
C.1.2	Changed Traceables in 17-10	118
C.1.3	Deleted Traceables in 17-10	120
C.1.4	Added Constraints in 17-10	120
C.1.5	Changed Constraints in 17-10	120
C.1.6	Deleted Constraints in 17-10	120
C.2	Constraint and Specification Item History of this document according to AUTOSAR Release 18-03	120
C.2.1	Added Traceables in 18-03	120
C.2.2	Changed Traceables in 18-03	122
C.2.3	Deleted Traceables in 18-03	123
C.2.4	Added Constraints in 18-03	124
C.2.5	Changed Constraints in 18-03	124
C.2.6	Deleted Constraints in 18-03	124
C.3	Constraint and Specification Item History of this document according to AUTOSAR Release 18-10	124
C.3.1	Added Traceables in 18-10	124
C.3.2	Changed Traceables in 18-10	125
C.3.3	Deleted Traceables in 18-10	126
C.3.4	Added Constraints in 18-10	126
C.3.5	Changed Constraints in 18-10	126
C.3.6	Deleted Constraints in 18-10	126
C.4	Constraint and Specification Item History of this document according to AUTOSAR Release 19-03	127
C.4.1	Added Traceables in R19-03	127
C.4.2	Changed Traceables in R19-03	127
C.4.3	Deleted Traceables in R19-03	128
C.4.4	Added Constraints in R19-03	128
C.4.5	Changed Constraints in R19-03	128
C.4.6	Deleted Constraints in R19-03	128
C.5	Constraint and Specification Item History of this document according to AUTOSAR Release 19-11	129
C.5.1	Added Traceables in R19-11	129
C.5.2	Changed Traceables in R19-11	130
C.5.3	Deleted Traceables in R19-11	133
C.5.4	Added Constraints in R19-11	133

C.5.5 Changed Constraints in R19-11 133
C.5.6 Deleted Constraints in R19-11 133

1 Introduction and functional overview

This document is the software specification of the [Execution Management](#) functional cluster within the [Adaptive Platform Foundation](#).

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

Chapter 7 describes how [Execution Management](#) concepts are realized within the [AUTOSAR Adaptive Platform](#).

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

1.1 What is Execution Management?

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

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

1.2 Interaction with AUTOSAR Runtime for Adaptive

The set of programming interfaces to the [Adaptive Applications](#) is called AUTOSAR Runtime for Adaptive (ARA). The interfaces that constitute ARA include those of [Execution Management](#) specified in Chapter 8. Note that interfaces which are exclusively accessed within the [AUTOSAR Adaptive Platform](#) are defined as inter-functional cluster (IFC) interfaces (see Appendix B), which are not part of ARA.

[Execution Management](#), in common with other [Applications](#) is assumed to be a process executed on a POSIX compliant operating system. [Execution Management](#) is responsible for initiating execution of the processes in all the Functional Clusters, Adaptive AUTOSAR Services, and user-level [Applications](#). The launching order is derived by [Execution Management](#) according to the specification defined in this document to ensure proper startup of the [AUTOSAR Adaptive Platform](#).

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

2 Acronyms and abbreviations

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

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

Term	Description
Application	An implementation that resolves a set of coherent functional requirements and is the result of functional development. An <code>Application</code> is the unit of delivery for <code>Machine</code> specific configuration and integration.
Executable	Part of an <code>Application</code> . It consists of executable code (with exactly one entry point) created at integration time that can be deployed and installed on a <code>Machine</code> . An <code>Application</code> may consist of one or more <code>Executables</code> , each of which can be deployed to different <code>Machines</code> .
Process	A <code>Process</code> is an instance of an <code>Executable</code> to be executed on a <code>Machine</code> and has a 1:1 association with the ARXML/Meta-Model element with the same name. This document also uses the term process (without a capital "P") to refer to the OS concept of a running process. Attention: <code>Process</code> != process. Hence each <code>Process</code> has at some time a related process but a process may not always have a related <code>Process</code> .
Reporting Process	A type of <code>Process</code> with an associated <code>Executable</code> where <code>reportingBehavior</code> is omitted ([TPS_MANI_01279]) or set to <code>reportsExecutionState</code> . Such a <code>Process</code> is expected to report its Execution State to <code>Execution Management</code> .
Non-reporting Process	A type of <code>Process</code> with an associated <code>Executable</code> where <code>reportingBehavior</code> set to <code>doesNotReportExecutionState</code> . Such a <code>Process</code> is not expected to report its Execution State to <code>Execution Management</code> .
Self-terminating Process	A type of <code>Process</code> that self initiate termination procedure, please note that for a standard <code>Process</code> this procedure is initiated by <code>Execution Management</code> .
Execution Dependency	Dependencies between <code>Executable</code> instances can be configured to define a sequence for starting and terminating them.
Execution Management	The element of the <code>AUTOSAR Adaptive Platform</code> responsible for the ordered startup and shutdown of the <code>AUTOSAR Adaptive Platform</code> and the <code>Applications</code> .
State Management	The element of <code>AUTOSAR Adaptive Platform</code> defining modes of operation. It allows flexible definition of functions which are active on the platform at any given time.

Function Group	A Function Group is a set of coherent Processes , which need to be controlled consistently. Depending on the state of the Function Group , Processes are started or terminated. Processes can belong to more than one Function Group State (but at exactly one Function Group). "MachineState" is a Function Group with a predefined name, which is mainly used to control Machine lifecycle and Processes of platform level Applications . Other Function Groups are sort of general purpose tools used (for example) to control Processes of user level Applications .
Function Group State	The state of a Function Group (except "MachineState"). It defines a set of active Applications for any certain situation. The set of Function Groups and their states is machine specific and is deployed as part of the Machine Manifest .
Machine State	The state of Function Group "MachineState" with some predefined states (Startup/Shutdown/Restart).
Time Determinism	The results of a calculation are guaranteed to be available before a given deadline.
Data Determinism	The results of a calculation only depend on the input data and are reproducible, assuming a given initial internal state.
Full Determinism	Combination of Time and Data Determinism.
Communication Management	A Functional Cluster within the Adaptive Platform Foundation
Execution Manifest	Manifest file to configure execution of an Adaptive Application . An Execution Manifest is created at integration time and deployed onto a Machine together with the Executable to which it is attached. It supports the integration of the Executable code and describes the configuration properties (startup parameters, resource group assignment etc.) of each Process , i.e. started instance of that Executable .
Machine Manifest	Manifest file to configure a Machine . The Machine Manifest holds all configuration information which cannot be assigned to a specific Executable or Process .
Operating System	Software responsible for managing Processes on a Machine and for providing an interface to hardware resources.
ResourceGroup	Configuration element to enable restrictions on resources uses by Adaptive Applications running in the group.
ExecutionClient	Adaptive Application interface to Execution Management .
DeterministicClient	Adaptive Application interface to Execution Management to support control of the process-internal cycle, a deterministic worker pool, activation time stamps and random numbers.
StateClient	State Management interface to Execution Management to support Function Group State and Machine State management.
Platform Health Management	A Functional Cluster within the Adaptive Platform Foundation
Recovery Action	Actions defined by the integrator to control Adaptive Application error recovery.
Process State	Lifecycle state of a Process
Service Instance Manifest	Manifest file to configure Service usage of an Adaptive Application .

Trusted Platform	An execution platform supporting a continuous chain of trust from boot through to application supporting authentication (that all code executed is from the claimed source) and integrity validation (that prevents tampered code/data from being executed).
------------------	--

Table 2.1: Technical Terms

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

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

Table 2.2: Glossary-defined Technical Terms

3 Related documentation

3.1 Input documents & related standards and norms

The main documents that serve as input for the specification of the [Execution Management](#) are:

- [1] Key words for use in RFCs to Indicate Requirement Levels
<http://www.ietf.org/rfc/rfc2119.txt>
- [2] Specification of Communication Management
AUTOSAR_SWS_CommunicationManagement
- [3] Glossary
AUTOSAR_TR_Glossary
- [4] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification
- [5] General Specification of Adaptive Platform
AUTOSAR_SWS_General
- [6] Requirements on Execution Management
AUTOSAR_RS_ExecutionManagement
- [7] Specification of Operating System Interface
AUTOSAR_SWS_OperatingSystemInterface
- [8] Specification of Persistency
AUTOSAR_SWS_Persistency
- [9] Specification of Platform Health Management for Adaptive Platform
AUTOSAR_SWS_PlatformHealthManagement
- [10] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology
- [11] Specification of State Management
AUTOSAR_SWS_StateManagement
- [12] Guidelines for using Adaptive Platform interfaces
AUTOSAR_EXP_AdaptivePlatformInterfacesGuidelines
- [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 Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004

[15] Explanation of Adaptive Platform Design
AUTOSAR_EXP_PlatformDesign

3.2 Further applicable specification

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

4 Constraints and assumptions

4.1 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 7.8 Resource Limitation and Section 7.9 Fault Tolerance – these sections have been expanded in this release but are not complete. In particular the contents will be expanded with more properties and formal requirements in the next release.

Section 6.1 details requirements from [Execution Management](#) Requirement Specification [6] 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

[Execution Management](#) is dependent on the Operating System Interface [7]. The OSI is used to control specific aspects of [Application](#) execution, for example, to set scheduling parameters or to execute an [Application](#).

[Execution Management](#) may dependent on the Operating System beyond the Operating System Interface [7], e.g to control the core affinity of processes (refer [7.8.3.2](#)).

[Execution Management](#) is dependent on the Persistency [8] functional cluster. Persistency is used to access persistent storage.

[Execution Management](#) is dependent on the Platform Health Management [9] functional cluster. The Platform Health Management Interfaces are used by [Execution Management](#) for notifying a state change of a [Process](#).

5.1 Protocol layer dependencies

None.

6 Requirements traceability

The following tables reference the requirements specified in [6] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_AP_00115]	Namespaces.	[SWS_EM_NA]
[RS_AP_00120]	Method and Function names.	[SWS_EM_02283] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02290] [SWS_EM_02291]
[RS_AP_00121]	Parameter names.	[SWS_EM_02283] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02291]
[RS_AP_00122]	Type names.	[SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284]
[RS_AP_00124]	Variable names.	[SWS_EM_NA]
[RS_AP_00127]	Usage of <code>ara::core</code> types.	[SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284]
[RS_AP_00128]	Error reporting.	[SWS_EM_02292]
[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_00132]	noexcept behavior of API functions	[SWS_EM_02283] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02290] [SWS_EM_02291] [SWS_EM_NA]
[RS_EM_00002]	Execution Management shall set-up one process for the execution of each <code>Process</code> .	[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 <code>Process</code> and groups of <code>Processes</code> .	[SWS_EM_02102] [SWS_EM_02103] [SWS_EM_02106] [SWS_EM_02107] [SWS_EM_02108] [SWS_EM_02109]
[RS_EM_00008]	Execution Management shall support the binding of all threads of a given <code>Process</code> to a specified set of processor cores.	[SWS_EM_02104]
[RS_EM_00009]	Execution Management shall ensure it is the sole entity starting <code>Processes</code> .	[SWS_EM_01030] [SWS_EM_01033]
[RS_EM_00010]	Execution Management shall support multiple instances of <code>Executables</code> .	[SWS_EM_01012] [SWS_EM_01072] [SWS_EM_01073] [SWS_EM_01074] [SWS_EM_01075] [SWS_EM_01076] [SWS_EM_01077] [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249]
[RS_EM_00011]	Execution Management shall support self-initiated graceful shutdown of <code>Processes</code> .	[SWS_EM_01404] [SWS_EM_01405]

Requirement	Description	Satisfied by
[RS_EM_00013]	Execution Management shall support configurable recovery actions.	[SWS_EM_01016] [SWS_EM_01062] [SWS_EM_01063] [SWS_EM_01064] [SWS_EM_02076] [SWS_EM_02077] [SWS_EM_02257] [SWS_EM_02261] [SWS_EM_02262]
[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_02304] [SWS_EM_02305] [SWS_EM_02306] [SWS_EM_02307] [SWS_EM_02308] [SWS_EM_02309] [SWS_EM_NA]
[RS_EM_00050]	Execution Management shall perform Machine-wide coordination of Processes.	[SWS_EM_NA]
[RS_EM_00051]	Execution Management shall provide APIs to the Process for configuring external trigger conditions for its activities.	[SWS_EM_NA]
[RS_EM_00052]	Execution Management shall provide APIs to the Process for configuring cyclic triggering of its activities.	[SWS_EM_01301] [SWS_EM_01302] [SWS_EM_01303] [SWS_EM_01304] [SWS_EM_01351] [SWS_EM_01352] [SWS_EM_01353] [SWS_EM_02201] [SWS_EM_02210] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02216]
[RS_EM_00053]	Execution Management shall provide APIs to the Process to support deterministic redundant execution of Processes.	[SWS_EM_01305] [SWS_EM_01306] [SWS_EM_01308] [SWS_EM_01310] [SWS_EM_01311] [SWS_EM_01312] [SWS_EM_01313] [SWS_EM_02202] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02220] [SWS_EM_02225] [SWS_EM_02230] [SWS_EM_02235]
[RS_EM_00100]	Execution Management shall support the ordered startup and shutdown of Processes.	[SWS_EM_01000] [SWS_EM_01001] [SWS_EM_01050] [SWS_EM_01051]
[RS_EM_00101]	Execution Management shall support State Management functionality.	[SWS_EM_01013] [SWS_EM_01023] [SWS_EM_01024] [SWS_EM_01025] [SWS_EM_01032] [SWS_EM_01033] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01107] [SWS_EM_01109] [SWS_EM_01110] [SWS_EM_02241] [SWS_EM_02242] [SWS_EM_02245] [SWS_EM_02250] [SWS_EM_02251] [SWS_EM_02253] [SWS_EM_02254] [SWS_EM_02255] [SWS_EM_02258] [SWS_EM_02259] [SWS_EM_02260]

Requirement	Description	Satisfied by
		[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] [SWS_EM_02275] [SWS_EM_02276] [SWS_EM_02277] [SWS_EM_02278] [SWS_EM_02279] [SWS_EM_02297] [SWS_EM_02298]
[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_01071] [SWS_EM_01401] [SWS_EM_01402] [SWS_EM_01403] [SWS_EM_01404] [SWS_EM_01405] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02030] [SWS_EM_02243] [SWS_EM_02244]
[RS_EM_00111]	Execution Management shall assist identification of Processes during Machine runtime.	[SWS_EM_NA]
[RS_EM_NA]		[SWS_EM_NA]

6.1 Not applicable requirements

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

7 Functional specification

[Execution Management](#) is a functional cluster contained in the [Adaptive Platform Foundation](#). [Execution Management](#) is responsible for all aspects of system execution management including platform initialization and startup / shutdown of [Applications](#).

[Execution Management](#) works in conjunction with the Operating System. In particular, [Execution Management](#) is responsible for configuring the Operating System to perform run-time scheduling and resource monitoring of [Applications](#).

This chapter describes the functional behavior of [Execution Management](#).

- Section [7.2](#) presents an introduction to key terms within [Execution Management](#) focusing on the relationship between [Application](#), [Executable](#), and [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 [Process](#) state transitions and startup / shutdown sequences.
- Section [7.5](#) covers several topics related to State Management within [Execution Management](#) including execution, [Machine](#) and [Function Group](#) state management.
- Section [7.6](#) describes how [Application](#) error recovery actions are specified during integration.
- Section [7.7](#) documents support provided by [Execution Management Deterministic](#) execution such that given the same input and internal state, a calculation will always produce the same output.
- Section [7.8](#) describes how [Execution Management](#) supports resource management including the limitation of usage of CPU and memory by an [Application](#).
- Section [7.9](#) 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 [7.10](#) covers the topic of [Trusted Platform](#), i.e. ensuring the integrity and authenticity of [Applications](#).

7.1 Functional Cluster Lifecycle

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 [7.5.2.3](#).

7.2 Technical Overview

This chapter presents a short summary of the relationship between [Application](#), [Executable](#), and [Process](#).

7.2.1 Application

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

[Application Executables](#) can be located on user level above the middleware or can implement functional clusters of the [AUTOSAR Adaptive Platform](#) (located on the level of the middleware), see [constr_1605] in [4].

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

7.2.2 Adaptive Application

An [Adaptive Application](#) is a specific type of [Application](#). The implementation of an [Adaptive Application](#) fully complies with the AUTOSAR specification,

i.e. it is restricted to the use of APIs standardized by AUTOSAR and needs to follow specific coding guidelines to allow reallocation between different implementations of the [AUTOSAR Adaptive Platform](#).

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

Figure 7.1 shows the different types of Applications.

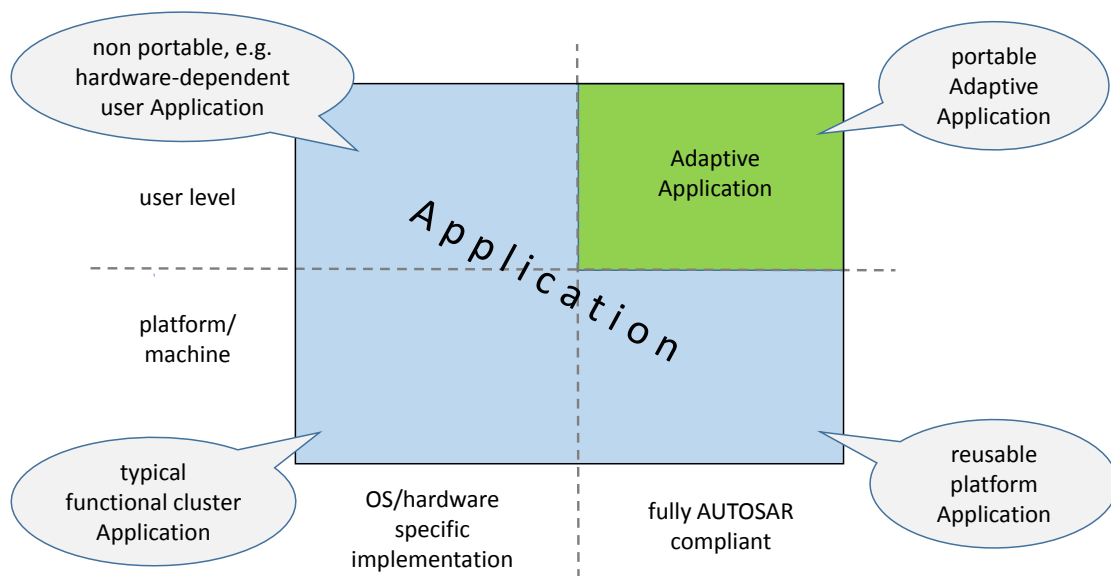


Figure 7.1: Types of Applications

An [Adaptive Application](#) is the result of functional development and is the unit of delivery for [Machine](#) specific configuration and integration. Some contracts (e.g. concerning used libraries) and [Service Interfaces](#) to interact with other [Adaptive Applications](#) need to be agreed on beforehand. For details see [10].

7.2.3 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].

The lifecycle of [Executables](#) usually consists of:

Process Step	Software	Meta Information
Development and Integration	Linked, configured and calibrated binary for deployment onto the target Machine . The binary might contain code which was generated at integration time.	Execution Manifest , see 7.2.5 and [4], and Service Instance Manifest (not used by Execution Management).

Deployment and Removal	Binary installed on the target <i>Machine</i> . Previous version (if any) removed.	Processed Manifests, stored in a platform-specific format which is efficiently readable at <i>Machine</i> startup.
Execution	<i>Process</i> started as instance of the binary.	The Execution Management uses contents of the Processed Manifests to start up and configure each <i>Process</i> individually.

Table 7.1: Executable Lifecycle

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

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

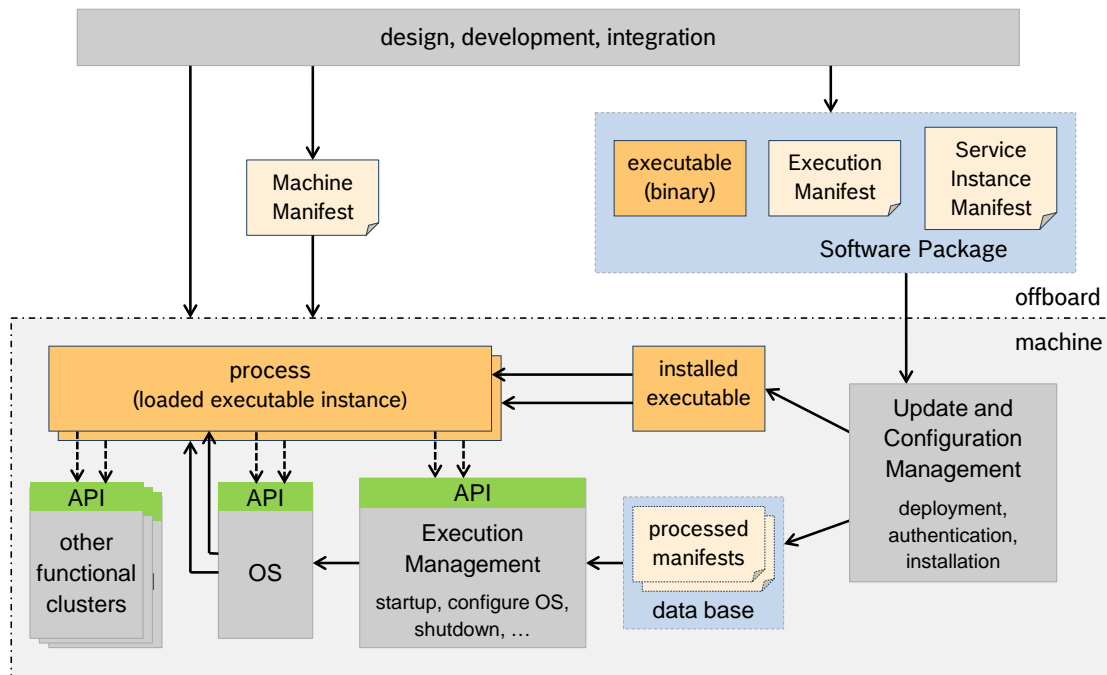


Figure 7.2: Executable Lifecycle from deployment to execution

7.2.4 Process

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

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

Remark: In this release of this document it is mostly assumed that [Processes](#) are self-contained, i.e. that they take care of controlling thread creation and scheduling by calling APIs of the Operating System Interface from within the code. [Execution Management](#) only starts and terminates the [Processes](#) and while the [Processes](#) are running, [Execution Management](#) only interacts with the [Processes](#) by providing [State Management](#) mechanisms (see [7.5](#)) or APIs to support Deterministic Execution (see [7.7.3](#)).

7.2.5 Execution Manifest

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

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

The [Execution Manifest](#) is bundled with the actual executable code in order to support the deployment of the executable code onto the [Machine](#).

Each instance of an [Executable](#) binary, i.e. each started [Process](#), is individually configurable, with the option to use a different configuration set per [Machine State](#) or per [Function Group State](#) (see Section [7.5](#) and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01014], [TPS_MANI_01015], [TPS_MANI_01059], [TPS_MANI_01017] and [TPS_MANI_01041]).

To perform its necessary actions, [Execution Management](#) imposes a number of requirements on the content of the [Execution Manifest](#).

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

7.2.6 Machine Manifest

The [Machine Manifest](#) is also created at integration time for a specific [Machine](#) and is deployed like [Execution Manifests](#) whenever its contents change. The [Machine Manifest](#) holds all configuration information which cannot be assigned to a specific [Executable](#) or its instances (the [Processes](#)), i.e. which is not already covered by an [Execution Manifest](#) or a [Service Instance Manifest](#).

The contents of a [Machine Manifest](#) includes the configuration of [Machine](#) properties and features (resources, safety, security, etc.), e.g. configured [Machine States](#) and [Function Group States](#), resource groups, access right groups, scheduler configuration, SOME/IP configuration, memory segmentation. For details see [4].

7.2.7 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 7.10.

[SWS_EM_01030]{DRAFT} Restriction of process creation right for Processes [`Execution Management` shall restrict the rights of `Processes` such that they cannot start other processes.](*RS_EM_00009*)

The mechanism by which the restriction of [SWS_EM_01030] is implementation-specific, but could be realized by configuring the process capability attribute mask at the time of process creation.

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

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

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

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 `ExecutionState` enumeration.

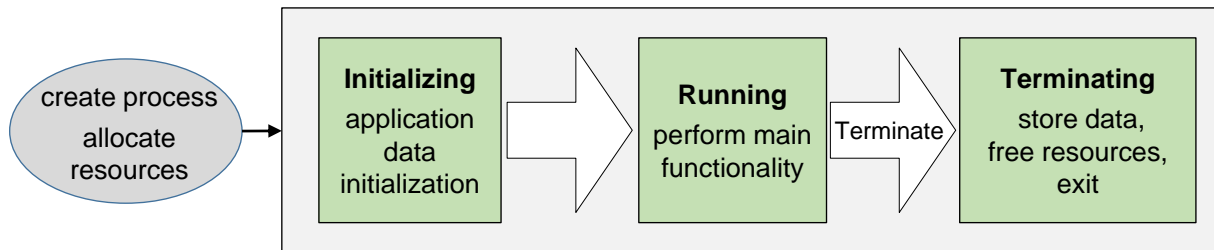


Figure 7.3: Execution States

Execution Management considers *Process* initialization complete when the state `kRunning` is reached whether this is achieved implicitly or explicitly through a *Process* reporting its *Execution State*.

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

[SWS_EM_01055] Initiation of Process termination [*Execution Management* shall initiate *Process* termination by sending the `SIGTERM` signal to the *Process*.] (*RS_EM_00103*)

Note that from the perspective of *Execution Management*, requirement [SWS_EM_01055] only requests the initiation of the steps necessary for termination.

On receipt of `SIGTERM`, a *Reporting Process* acknowledges the request (by reporting `kTerminating` to *Execution Management* using the `ExecutionClient::ReportExecutionState` interface) and then commences the actual termination. A *Reporting Process* initiates self-termination by reporting the `kTerminating` state to *Execution Management*.

[SWS_EM_01071]{DRAFT} Premature Termination of a Reporting Process [*Execution Management* shall consider the termination of a *Reporting Process* without prior reporting of `kTerminating` as an error. The error reaction is implementation specific.] (*RS_EM_00103*)

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

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

Details on the response to “fault” error-codes, e.g. a non-zero exit code, will be defined in Section 7.9 in a future release of this document.

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

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

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

Note that in [SWS_EM_01065], *Process* termination timeout monitoring applies to all termination sequences including self-termination (*Process* reports *Execution State* Terminating without request by *Execution Management*) and termination requested by *Execution Management*.

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.

[SWS_EM_01402] Implicit Running Process State [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.] (*RS_EM_00103*)

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 *Resource-Groups* can be used (see Section 7.8).

[SWS_EM_01403]{DRAFT} Reporting Non-reporting Process [Execution Management shall consider Execution State reporting from a Non-reporting Process as an error. The error reaction is implementation specific.](RS_EM_00103)

7.4.2 Process States

Process States characterize the lifecycle of a *Process* from the point of view of *Execution Management*. In other words, they represent *Execution Management* internal tracking of the *Execution States* (see Section 7.4.1). Note that each *Process* is independent and therefore has its own *Process State*.

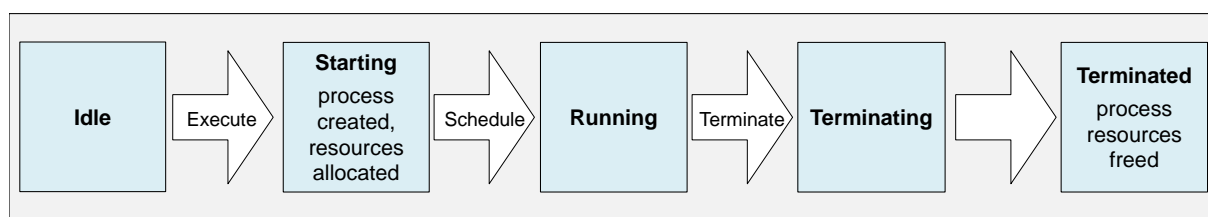


Figure 7.4: Process Lifecycle

[SWS_EM_01401] Process Self Reporting [The AdaptivePlatform implementation shall only allow a *Process* to report its own *ExecutionState*.](RS_EM_00103)

[SWS_EM_01002] Idle Process State [The *Idle* Process State shall be the Process State prior to creation of the *Process* and to resource allocation.](RS_EM_00103)

[SWS_EM_01003] Starting Process State [The *Starting* Process State shall apply when the *Process* has been created and resources have been allocated.](RS_EM_00103)

[SWS_EM_01004] Running Process State of Reporting Processes [The *Running* Process State shall apply to a *Reporting Process* after it has reported *kRunning* Execution State to *Execution Management*.](RS_EM_00103)

[SWS_EM_01404] Terminating Process State after Termination Request [The *Terminating Process State* shall apply when *Execution Management* sent SIGTERM signal to the *Process*.](RS_EM_00103, RS_EM_00011)

[SWS_EM_01405] Terminating Process State after Terminating Report [The *Terminating Process State* shall apply when the *Reporting Process* has decided to self-terminate and informed *Execution Management* by reporting *kTerminating* Execution State.](RS_EM_00103, RS_EM_00011)

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

[SWS_EM_01006] Terminated Process State [The *Terminated* Process State shall apply after the *Process* has terminated and the *Process* resources have been freed.](RS_EM_00103)

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].

7.4.3 Startup and Termination

7.4.3.1 Ordering

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

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

When considering dependencies on *Services* it is tempting to use *Execution Dependencies*. However general *Service Discovery* is a better mechanism as services can go ON or OFF at any time and the *Processes* are expected to cope with this situation. Please note that *Execution Dependencies* may be used to reduce service discovery delays.

Execution Dependencies, see [TPS_MANI_01041] and [constr_1606], are configured in the *Execution Manifests*.

Example 7.1

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

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

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 [During termination of a [Process](#), [Execution Management](#) shall respect [Execution Dependencies](#) by ensuring that any [Processes](#) upon which the [Process](#) to be terminated depends are not terminated before termination of the [Process](#).] ([RS_EM_00100](#))

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.

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

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

7.4.3.2 Arguments

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

[SWS_EM_01012] Process Argument Passing [At the initiation of startup of a [Process](#), the aggregated [StartupOptions](#) of the [StartupConfig](#) referenced by the [StateDependentStartupConfig](#) shall be passed to the [Process](#) by [Execution Management](#) based on [\[SWS_EM_01072\]](#), [\[SWS_EM_01073\]](#), [\[SWS_EM_01074\]](#), [\[SWS_EM_01075\]](#), [\[SWS_EM_01076\]](#) and [\[SWS_EM_01077\]](#).] ([RS_EM_00010](#))

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 on the command-line passed by `Execution Management` is the name of the `Executable`.

[SWS_EM_01072] Process Argument Zero [Argument 0 shall be set to name of the `Executable`.] (*RS_EM_00010*)

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

[SWS_EM_01073] Simple Arguments [For each aggregated `StartupOption` at position n with `StartupOption.optionKind = commandLineSimpleForm` the n th argument shall be `StartupOption.optionArgument`.] (*RS_EM_00010*)

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

[SWS_EM_01074] Short form arguments with option value [For each aggregated `StartupOption` at position n with `StartupOption.optionKind = commandLineShortForm` and with multiplicity of `StartupOption.optionArgument = 1` the n th argument shall be '-' + `StartupOption.optionName` + '_' + `StartupOption.optionArgument`] (*RS_EM_00010*)

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

[SWS_EM_01075] Short form Arguments without option value [For each aggregated `StartupOption` at position n with `StartupOption.optionKind = commandLineShortForm` and with multiplicity of `StartupOption.optionArgument = 0` the n th argument shall be '-' + `StartupOption.optionName`] (*RS_EM_00010*)

`Execution Management` supports long form arguments which are typically more meaningful to the user than short-form arguments. To distinguish long form arguments from short form the former begin with a double dash (--) which is not included in the `StartupOption.optionName`.

[SWS_EM_01076] Long form Arguments with option value [For each aggregated `StartupOption` at position n with `StartupOption.optionKind = commandLineLongForm` and with multiplicity of `StartupOption.optionArgument = 1` the n th argument shall be '--' + `StartupOption.optionName` + '=' + `StartupOption.optionArgument`] (*RS_EM_00010*)

[SWS_EM_01077] Long form Arguments without option value [For each aggregated `StartupOption` at position n with `StartupOption.optionKind = commandLineLongForm` and with multiplicity of `StartupOption.optionArgument = 0` the n th argument shall be '--' + `StartupOption.optionName`] (*RS_EM_00010*)

7.4.3.3 Environment Variables

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

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

[SWS_EM_02247] Machine specific Environment Variables [`Execution Management` shall prepare environment variables based on the configuration from `Machine.environmentVariable` and pass them during a `Process` start.](*RS_EM_00010, RS_AP_00130*)

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

[SWS_EM_02249] Missing value from Environment Variable definition [Whenever `Execution Management` finds environment variable definition, that have `TagWithOptionalValue.value` missing, it should use empty string as a value for this environment variable.](*RS_EM_00010, RS_AP_00130*)

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

7.4.4 Startup Sequence

When the `Machine` is started, `Execution Management` is launched as the `Machine`'s first process. Other functional clusters and platform-level `Applications` of the `Adaptive Platform Foundation` and `Adaptive Platform Services` are then launched by `Execution Management`. After the `Adaptive Platform Foundation` and `Adaptive Platform Services` are up and running, `Execution Management` continues to launch user-level `Applications`.

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

[SWS_EM_01000] Startup order [The startup order of the platform-level `Processes` shall be determined by `Execution Management` based on `Machine Manifest` and `Execution Manifest` information.](*RS_EM_00100*)

Please see Section 7.2.5.

Figure 7.5 shows the overall startup sequence.

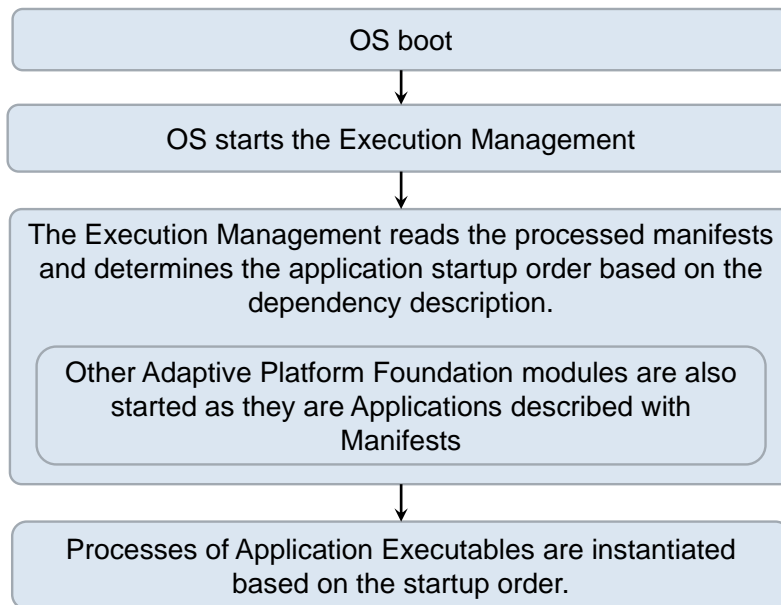


Figure 7.5: Startup sequence

7.4.4.1 Execution Dependency

Execution Management provides support to the AUTOSAR Adaptive Platform for ordered startup and shutdown of Applications. This ensures that Applications are started before dependent Applications use the services that they provide and, likewise, that Applications are shutdown only when their provided services are no longer required.

The Execution Dependencies, see [TPS_MANI_01041] and [constr_1606], are configured in the Execution Manifests, which is created at integration time based on information provided by the Application developer.

User-level Applications are expected to use the service discovery mechanisms of Communication Management as the primary mechanism for execution sequencing as this is supported both within a Machine and across Machine boundaries. Thus user-level applications should not rely on Execution Dependencies unless strictly necessary. Which Processes are running depends on the current Function Group States, including the Machine State, see Section 7.5. The integrator should ensure that all service dependencies are mapped to State Management configuration, i.e. that all dependent Processes are running when needed.

In real life, specifying a simple dependency to a Process might not be sufficient to ensure that the depending service is actually provided. Since some Processes shall reach a certain Execution State (see Section 7.4.1) 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* reports *Running* state to the EM.

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

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

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

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

The *version* information within the *Execution Manifest* is available for use during integration to ensure that a consuming *Executable* and its required services are compatible with the producing *Executable* and its provided services. This also applies to the *Processes* which are instantiated from these *Executables*. An example for the definition of the version information attached to several *Executables* can be found in Listing Figure 7.1.

Listing 7.1: Example for Executable versions

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

```

<EXECUTABLE>
  <SHORT-NAME>SensorFusion</SHORT-NAME>
  <VERSION>1.0.2</VERSION>
</EXECUTABLE>
</ELEMENTS>
</AR-PACKAGE>
    
```

An example for the definition of the [Execution Dependency](#) information can be found in Listing Figure 7.2

Listing 7.2: Example for Executable dependency

```

<PROCESS>
  <SHORT-NAME>SensorFusion</SHORT-NAME>
  <EXECUTABLE-REF DEST="EXECUTABLE"/>/Executables/SensorFusion</EXECUTABLE-REF>
  <MODE-DEPENDENT-STARTUP-CONFIGS>
    <MODE-DEPENDENT-STARTUP-CONFIG>
      <EXECUTION-DEPENDENCIES>
        <EXECUTION-DEPENDENCY>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-DECLARATION-GROUP-PROTOTYPE"/>/Processes/RadarSensorVR/ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION"/>/ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-MODE-DECLARATION-REF>
          </PROCESS-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-DECLARATION-GROUP-PROTOTYPE"/>/Processes/RadarSensorVL/ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION"/>/ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-MODE-DECLARATION-REF>
          </PROCESS-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <PROCESS-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-DECLARATION-GROUP-PROTOTYPE"/>/Processes/Diag/ProcessStateMachine</CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION"/>/ModeDeclarationGroups/ProcessStateMachine/Running</TARGET-MODE-DECLARATION-REF>
          </PROCESS-MODE-IREF>
        </EXECUTION-DEPENDENCY>
      </EXECUTION-DEPENDENCIES>
      <STARTUP-CONFIG-REF DEST="STARTUP-CONFIG"/>/StartupConfigSets/StartupConfigSet_AA/SensorFusion_Startup</STARTUP-CONFIG-REF>
    </MODE-DEPENDENT-STARTUP-CONFIG>
  </MODE-DEPENDENT-STARTUP-CONFIGS>
</PROCESS>
    
```

```
</MODE-DEPENDENT-STARTUP-CONFIGS>  
</PROCESS>
```

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

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

Example 7.3

Let assume that Process “A” depends on the *Running Process State* of a Process “B”. At a Machine State transition, Process “A” shall be started, because it references the new Machine State. However, Process “B” does not reference that Machine State, so it is not started. Due to the Execution Dependency between the two Processes, Process “A” would never start running in the new Machine State because it waits forever for Process “B”. This is considered to be a configuration error and shall also cause run time error.

Please note that 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]. This is done on purpose, as this kind of dependencies will introduce hidden dependencies between Function Groups and they will not be visible to State Management. If dependencies between Function Groups needs to be expressed (e.g. mapping software could have dependency on GPS software), then this should be done inside State Management. For more information see [11].

Unlike a Reporting Process, a Non-reporting Process is in Execution 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 do not work in combination with 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 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**. This approach can also be used to communicate a Health State to Platform Health Management.

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

Four different states are relevant for [Execution Management](#):

Execution State – An Execution States characterizes the internal lifecycle of each started [Process](#), see Section [7.4.1](#)

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

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

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

[Function Group States](#) (including [Machine States](#)), define the current set of running [Processes](#). Each [Application](#) can declare in its [Execution Manifests](#) in which [Function Group States](#) its [Processes](#) shall be running. A [ModeDeclaration](#) for each required [Machine State](#) has to be defined in the [Machine Manifest](#) [[constr_1687](#)] [[TPS_MANI_03194](#)].

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]{DRAFT} Machine States configuration [Execution Management shall obtain configuration of the Function Group “MachineState” from Machine Manifest and set-up Machine States management.] (RS_EM_00101)

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

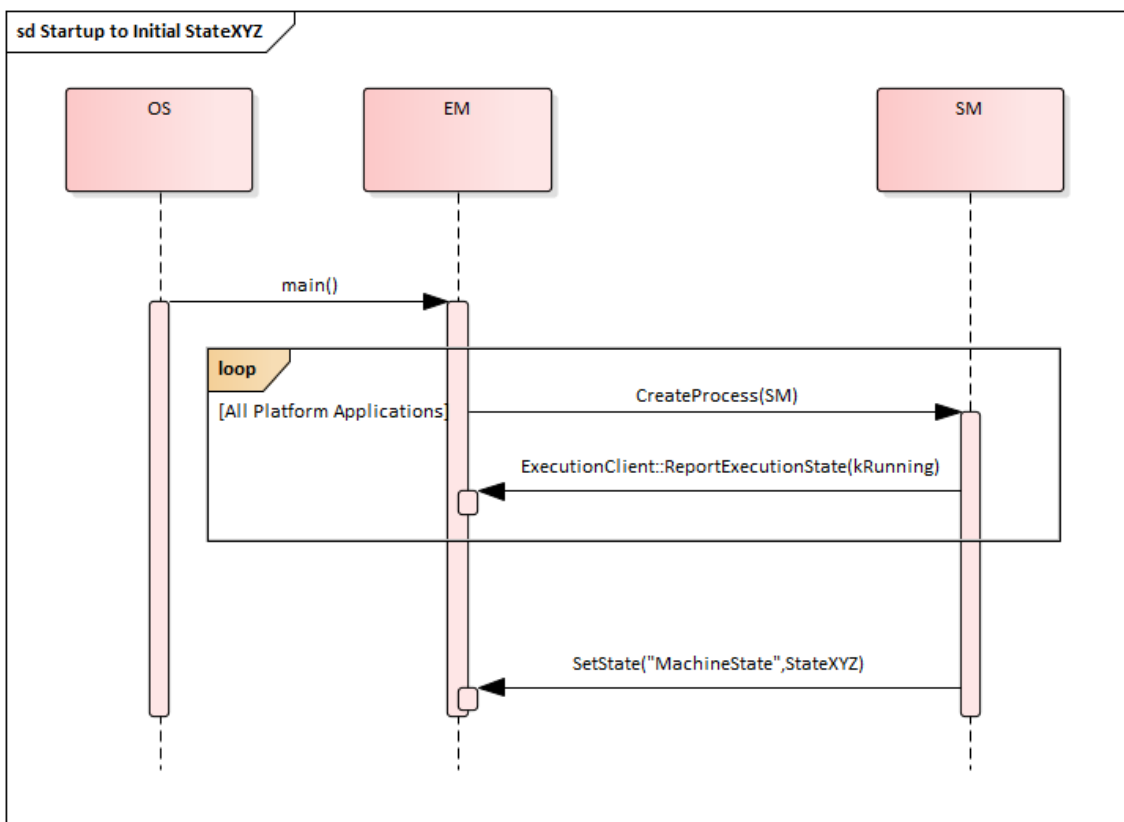


Figure 7.6: Start-up Sequence – from Startup to initial running state StateXYZ

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

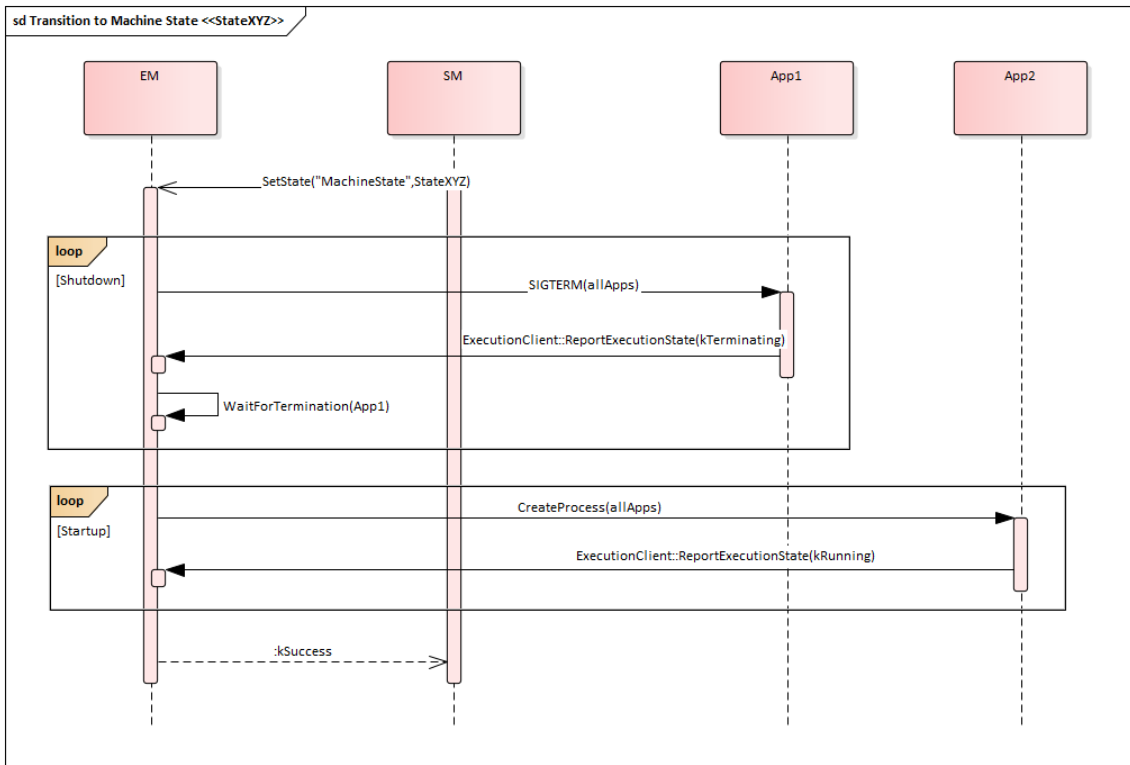


Figure 7.7: State Change Sequence – Transition to machine state StateXYZ

7.5.2.1 Startup

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

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

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

[SWS_EM_02241]{DRAFT} Machine State Startup Completion [Upon completion of initial (self initiated) Machine State transition to the Startup state, Execution Management shall notify State Management that the Startup state of Machine State has been reached.](RS_EM_00101)

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

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

7.5.2.2 Shutdown

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

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

A request to `Execution Management` to change the current `Machine State` to `Shutdown` is handled the same as any other `Function Group` state change request. From the point of view of `Execution Management` all `Function Groups` are independent and therefore changes to them, can be applied without any side effects. However, from the point of view of `State Management`, where knowledge of the dependencies between different `Function Groups` exist this may not be true. AUTOSAR assumes that `State Management` will requests `Machine State Shutdown` when it's valid to do so; see [11] for advice on how to orchestrate shutdown of the `Machine`.

As mentioned in Section 7.5.2.1 AUTOSAR assumes that `State Management` will be configured to run in `Shutdown`. State transition is not a trivial system change and it can fail for a number of reasons - in which case `State Management` should remain alive so you can report an error and wait for further instructions. Please note that very purpose of this state is to shutdown `Machine` (this includes `State Management`) in a clean manner. Unfortunately this means that at some point `State Management` will no longer be available to report errors and subsequent errors should be handled through implementation specific mechanisms.

7.5.2.3 Restart

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

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

From the point of view of [Execution Management](#), the [Restart](#) state of a [Function Group](#) with name "MachineState" is very similar to a [Shutdown](#) state. For the reasons mentioned in Section 7.5.2.2, a state transition to [Restart](#) is handled the same as any other [Function Group](#) state transition; please see [11] for advice on how to orchestrate restart of the [Machine](#).

As mentioned in Section 7.5.2.1 AUTOSAR assumes that [State Management](#) will be configured to run in [Restart](#). The reasons for doing so are the same as for Section 7.5.2.2.

7.5.3 Function Group State

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

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

The [Function Groups](#) mechanism is very flexible and is intended as a tool used to start and stop [Processes](#) of [Applications](#). System integrator can assign [Processes](#) to a [Function Group State](#) and then request it by [State Management](#). For details on state change management see Section 7.5.5.

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 functionally coherent user level [Applications](#). Please note that this doesn't mean that all [Processes](#) of platform level [Applications](#) has to be controlled by [Machine States](#).

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

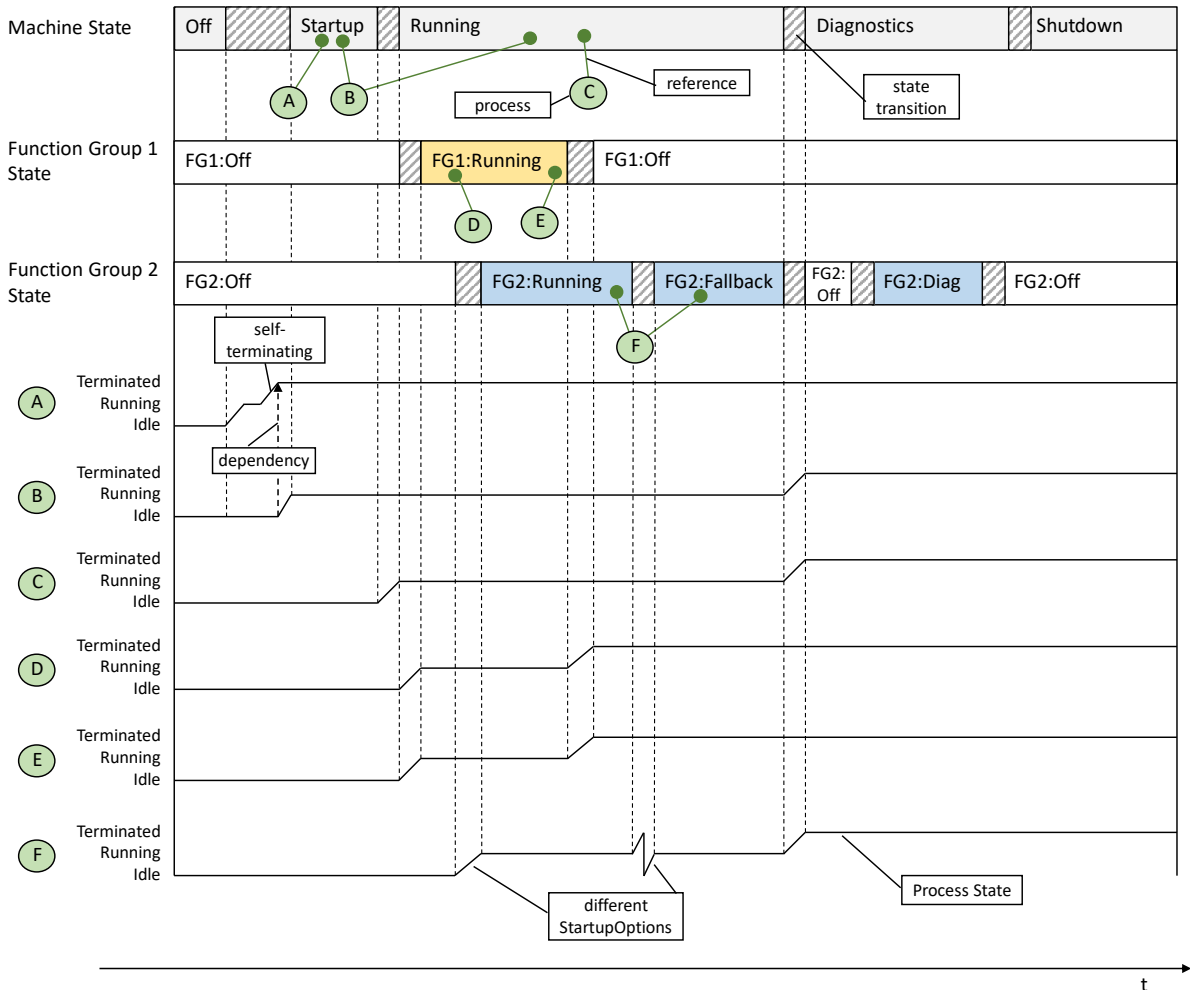


Figure 7.8: State dependent process control

- **Process A** references the Machine State Startup. It is a Self-terminating Process, i.e. it terminates after executing once.
- **Process B** references Machine States Startup and Running. It depends on the termination of Process A, i.e. an Execution Dependency has been configured, as described in Section 7.4.4.1
- **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.

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

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

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

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

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

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

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

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

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

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

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

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

The arbitrary state change sequence as shown in [Figure 7.7](#) applies to state changes of any [Function Group](#) - just replace "MachineState" by the name of the [Function Group](#). On receipt of the state change request, [Execution Management](#) terminates no longer needed [Processes](#) and then starts [Processes](#) active in the new [Function Group State](#) before confirming the state change to [State Management](#). For details see [Section 7.5.5](#).

7.5.4 State Interaction

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

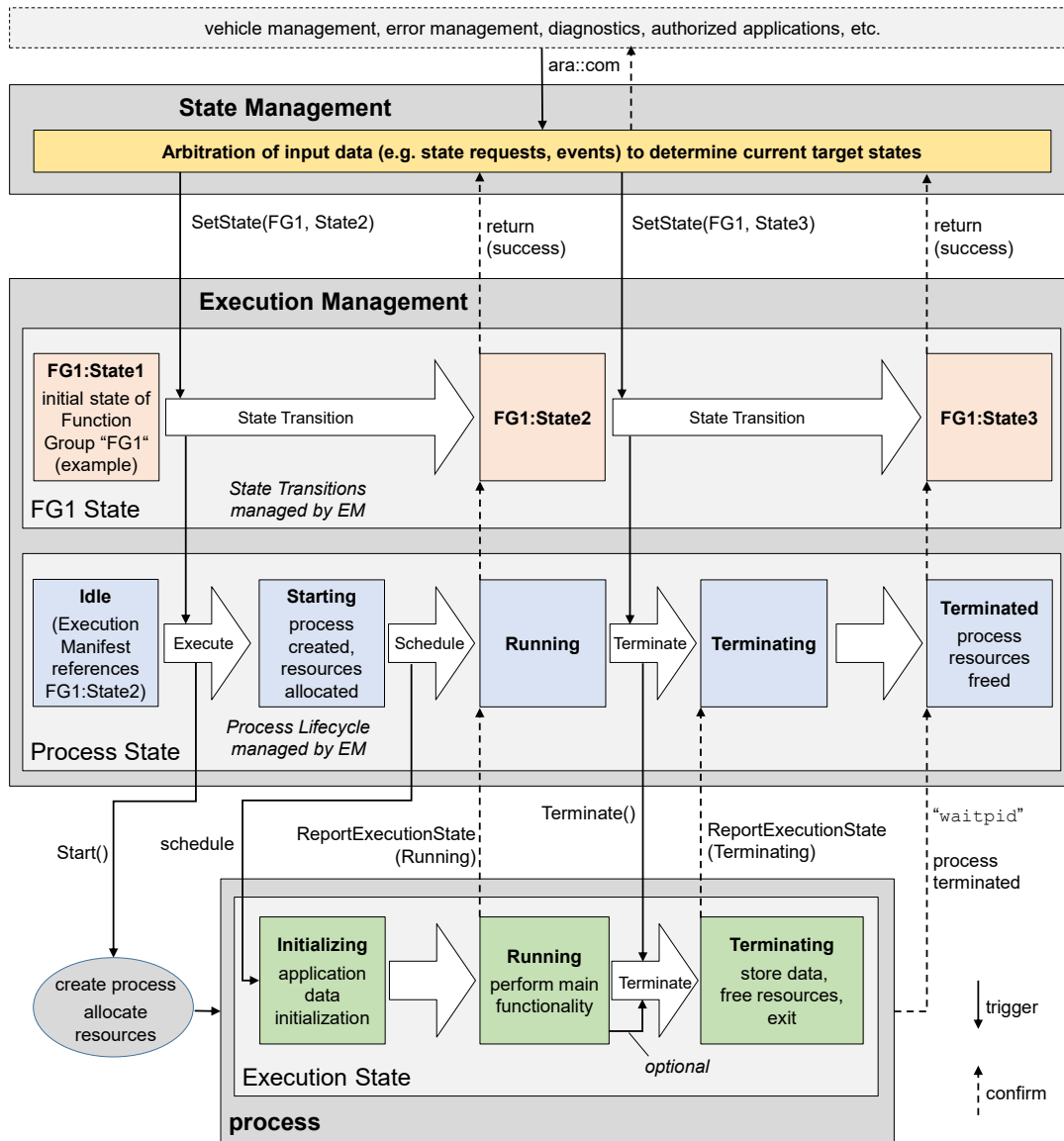


Figure 7.9: Interaction between states

7.5.5 State Transition

State Management can request to change one or several Function Group States (including the Machine State), using API described in Section 8.2.5. StateClient::SetState allows State Management to request several Function Group State changes in parallel. If Machine State change is required, the name of the Function Group passed shall be: "MachineState".

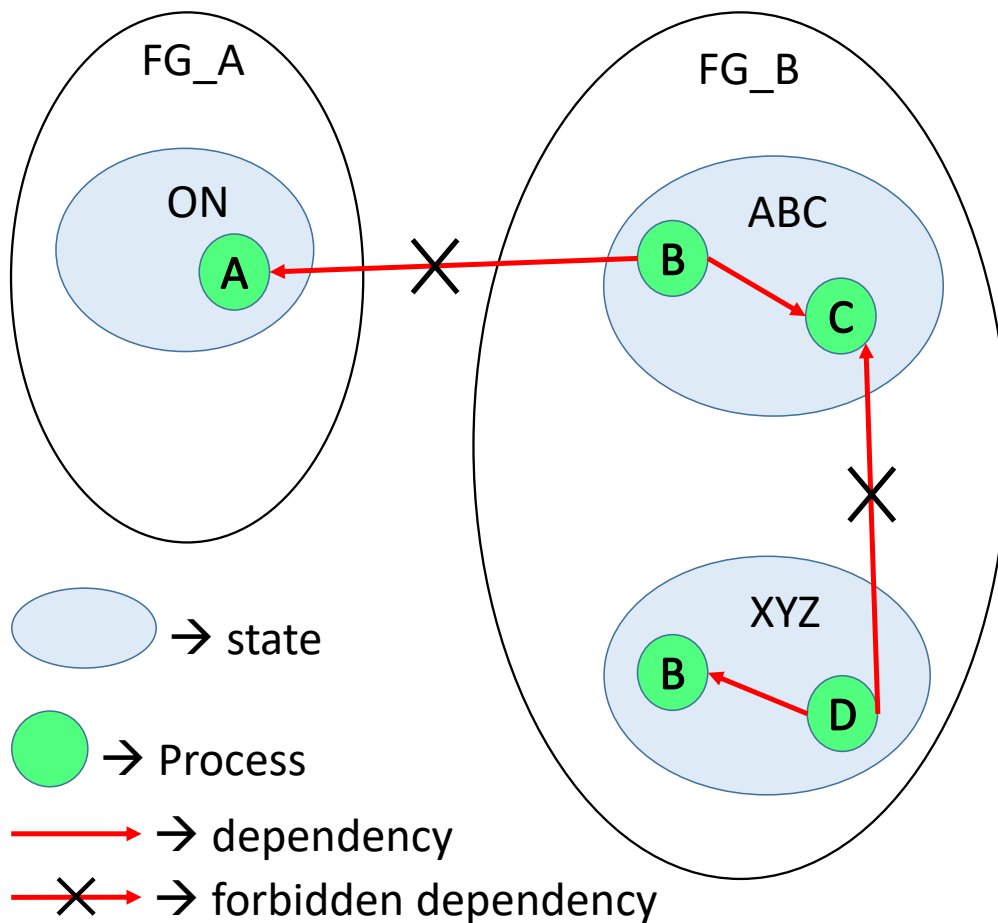


Figure 7.10: Example configuration for state transition

Before we specify how internals of a state transition works, let's consider an example configuration illustrated in figure Figure 7.10. As we can see [Execution Dependencies](#) that spans outside of a [Function Group](#) and moreover of a single [Function Group State](#) are forbidden. The dependency from [Process B](#) (inside [Function Group FG_B](#)) to [Process A](#) (inside [Function Group FG_A](#)) is forbidden, as it would introduce hidden dependencies between [Function Groups](#) that are not visible to [State Management](#). If system configuration requires this kind of dependencies, please see [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 chapter Section 7.4.4.1 ([\[SWS_EM_01001\]](#) and [\[constr_1689\]](#)).

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

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

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

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

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

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

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

Please see Section 7.4.1 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]{DRAFT} State transition - termination behavior [On state transition `Execution Management` shall terminate all `Processes` that references the `CurrentState` in its `Execution Manifest`, but don't references the `RequestedState` and have `Process State` different than [Idle or Terminated].] (*RS_EM_00101*)

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

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

Execution Management monitors the time required by each *Process* to terminate. The default value of the *Process* termination timeout is defined by the system integrator in the *Machine Manifest*, see [TPS_MANI_03151]. This value may be overwritten in the startup configuration of individual *Processes* by defining the termination timeout parameter in the *Execution Manifest*, see [TPS_MANI_01278].

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

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

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

[SWS_EM_02258]{DRAFT} State transition - Process termination timeout reporting [When the termination of a *Process* resulted in the timeout, *Execution Management* shall perform following actions:

- Report error code back to *State Management* to indicate that the *State* change request cannot be fulfilled.

] (*RS_EM_00101*)

Note that in [SWS_EM_01065], [SWS_EM_02255] and n [SWS_EM_02258], *Process* termination timeout monitoring applies to all termination sequences including self-termination.

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

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].

[SWS_EM_02253]{DRAFT} State transition - Process start-up timeout monitoring [*Execution Management* shall monitor the time required by the *Process* to start-up (the time between *Execution Management* requesting process creation from the operating system and the *Process* successfully reporting the *Running Process State*).] (*RS_EM_00101*)

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

[In case a `Process` start-up timeout occurred, `Execution Management` shall request `Process` termination. If `Process` doesn't terminate as requested, `Execution Management` shall request the `Operating System` to terminate the underlying process. After the process is terminated, the `Process` shall be restarted a `NumberOfRestartAttempts` number of times until success or final fail.]([RS_EM_00101](#))

[SWS_EM_02259]{DRAFT} State transition - Process start-up timeout reporting

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

- Report error code back to `State Management` to indicate that the `State` change request cannot be fulfilled.

]([RS_EM_00101](#))

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

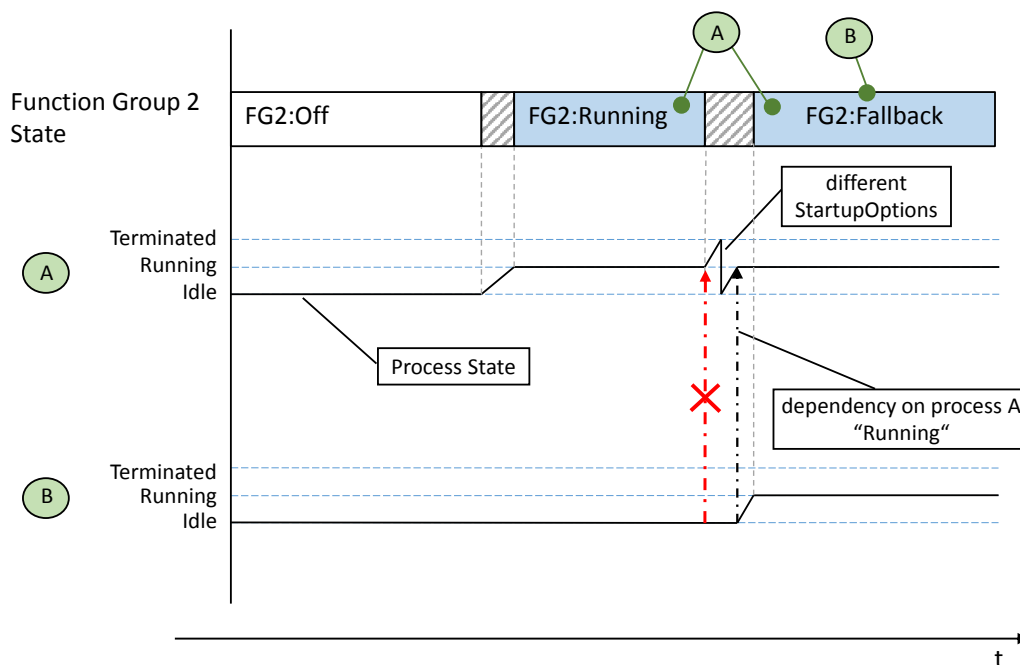


Figure 7.11: Dependency resolution during state change

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

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.

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.

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

[SWS_EM_02297]{DRAFT} StateClient usage restriction [AdaptivePlatform implementation shall only allow State Management to use interface implemented in StateClient class.] (RS_EM_00101)

If not protected StateClient can be used to destabilise Machine, see Section 8.2.5 for more details.

7.6 Application Recovery Actions

7.6.1 Overview

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

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

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

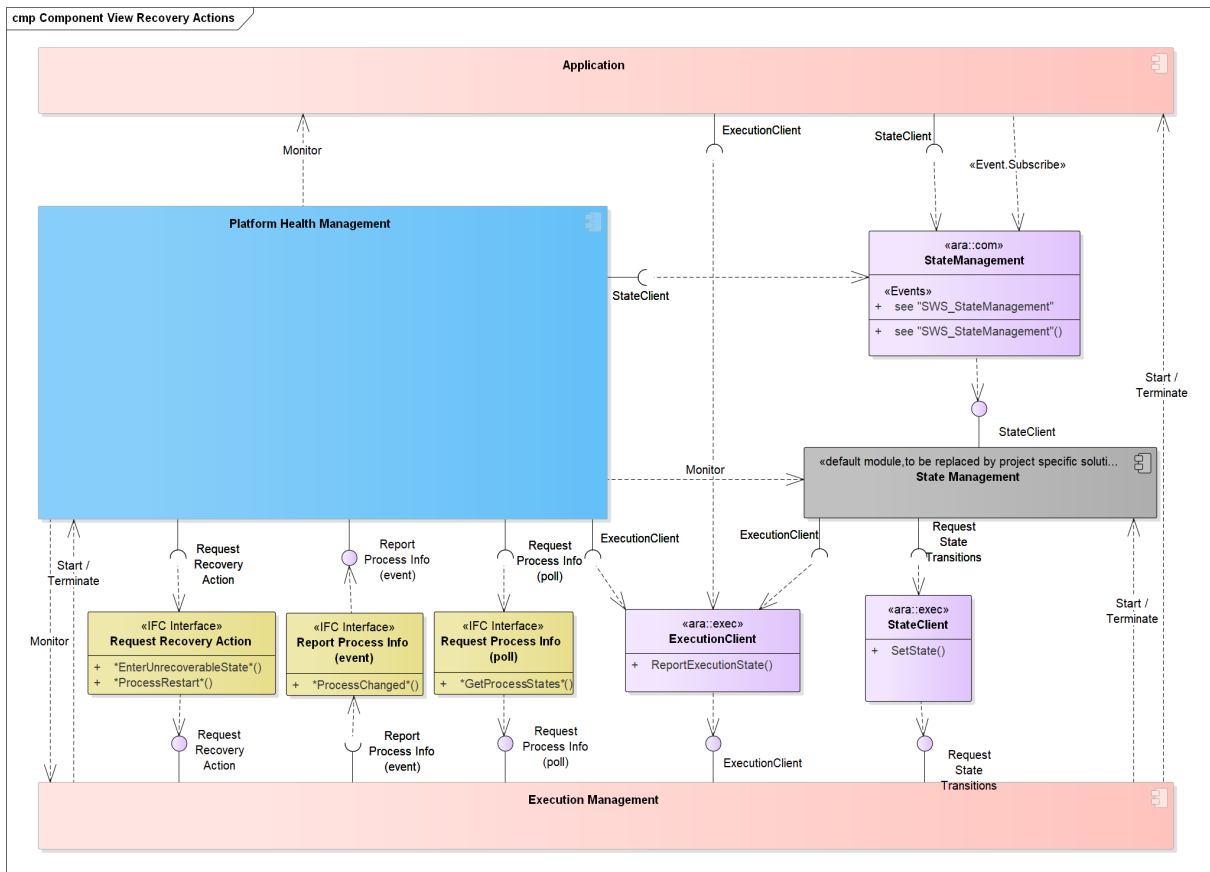


Figure 7.12: Adaptive Platform - Recovery Action Architecture

[SWS_EM_02257]{DRAFT} **Recovery Action API Security** [The AdaptivePlatform implementation shall only allow Platform Health Management to use the Process State Information and RecoveryAction APIs.] (RS_EM_00013)

7.6.2 Process State Information

7.6.2.1 Get Process States Information

[SWS_EM_02076]{DRAFT} Get Process States Information [Execution Management shall provide an inter functional cluster interface for Platform Health Management to receive a list of all currently running Processes.] (RS_EM_00013)

7.6.2.2 Process State Transition Event

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

7.6.3 Recovery Actions

7.6.3.1 Process Restart

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

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

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

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

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

7.6.3.2 Enter Unrecoverable State

[SWS_EM_02261]{DRAFT} Enter Unrecoverable State [Execution Management shall provide an inter functional cluster interface to force Execution Management to switch to the Unrecoverable State on the request from Platform Health Management.] (RS_EM_00013)

[SWS_EM_02262]{DRAFT} Enter Unrecoverable State Behavior [The Machine will never return from this Unrecoverable State.] ([RS_EM_00013](#))

The implementation of an Enter Unrecoverable State request could trigger the following actions:

- call a "pre-cleanup action" to inform others about the switch to the Unrecoverable State
- shutdown all processes
- call a "post-cleanup action" to set the machine in a physically known state, the machine will NOT return from this State. However as explained in the limitation above this is an example and NOT prescriptive.

Examples:

pre-cleanup action (optional):

- information another machine about the status
 - for error reporting to the driver via the cluster display (Level 1,2)
 - to activate a backup functionality (Level 4,5)

post-cleanup action (mandatory):

- to set the machine in a physically known state (switch the HW in a safe and stable state)
- the [Adaptive Application](#) machine will NOT return from this State

7.7 Deterministic Execution

7.7.1 Determinism

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

In the context of [Execution Management](#), the term “calculation” can apply to execution of a thread, a [Process](#), or a group of [Processes](#). The calculation can be event-driven or cyclic; i.e. time-driven.

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

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

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

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

Expected use cases of the [AUTOSAR Adaptive Platform](#) where such determinism is required include:

- Software Lockstep: To execute ASIL C/D applications with high computing performance demands, specific measures, such as software lockstep are required, due to high transient hardware error rates of high performance microprocessors. Software lockstep is a technique where the calculation is done redundantly through two different execution paths and the results are compared. To make the redundant calculations comparable, software lockstep requires a fully deterministic calculation. For details see [7.7.2](#).
- Reuse of verified software: The deterministic subsystem shows the same behavior on different platforms which satisfy the performance and resource needs of the subsystem, regardless of other differences in each environment, such as existence of unrelated applications. Examples include the different development and simulation platforms. Due to reproducible functional behavior, many results of testing, configuration and calibration of the subsystem are valid in each environment where the subsystem is deployed on and don't need to be repeated.

7.7.1.1 Time Determinism

Each time a calculation is started, its results are guaranteed to be available before a specified deadline. To achieve this, sufficient and guaranteed computing resources (processor time, memory, service response times etc.) should be assigned to the software entities that perform the calculation. For more information on resources see chapter 7.8.

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

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

Fully-Deterministic behavior additionally requires Data Determinism, however in many cases Time Determinism is sufficient.

7.7.1.2 Data Determinism

For Data Determinism, each time a calculation is started, its results only depend on the input data. For a specific sequence of input data, the results always need to be exactly the same, assuming the same initial internal state.

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

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

7.7.1.3 Full Determinism

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

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

Non-deterministic behavior may arise from different reasons; for example insufficient computing resources, or uncoordinated access of data, potentially by multiple threads running on multiple processor cores. The order in which the threads access such data will affect the result, which makes it non-deterministic (“race condition”).

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

7.7.2 Redundant Deterministic Execution

As explained in [7.7.1](#), future systems need high computing performance in combination with high ASIL safety goals. In this chapter we specify mechanisms which support deterministic multithread execution to support high performance software lockstep solutions. Here are some additional rationales behind it:

- Safety goals for Highly Automated Driving (HAD) systems can be up to ASIL D.
- High Performance Computing (HPC) demands can only be met by non automotive-grade, e.g. consumer electronics (CE), microprocessors, which have high transient hardware error rates compared to automotive-grade microcontrollers. Most likely no such microprocessor is available for ASIL above B, at least for the parts relevant to the design.
- To deal with high error rates, ASIL C/D HAD applications require specific measures, in particular software lockstep, where execution is done redundantly through two different paths and the result is compared to detect errors.
- To make these redundant calculations comparable, software lockstep requires a fully deterministic calculation as defined in [7.7.1.3](#).
- To meet HPC demands, highly predictable and reliable multi-threading must be supported

Figure [7.13](#) shows a simplified example for a possible software lockstep architecture.

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

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

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

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

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

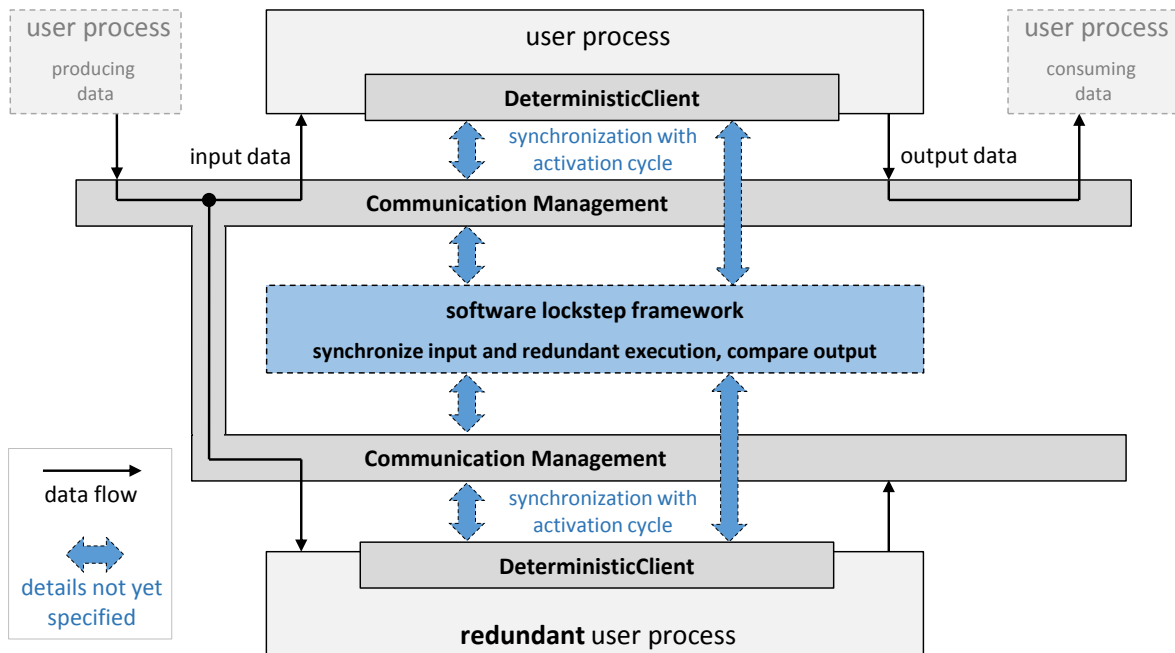


Figure 7.13: Software Lockstep in a typical data flow processing

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

Figure 7.14 zooms into one of the redundantly executed *Processes*.

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

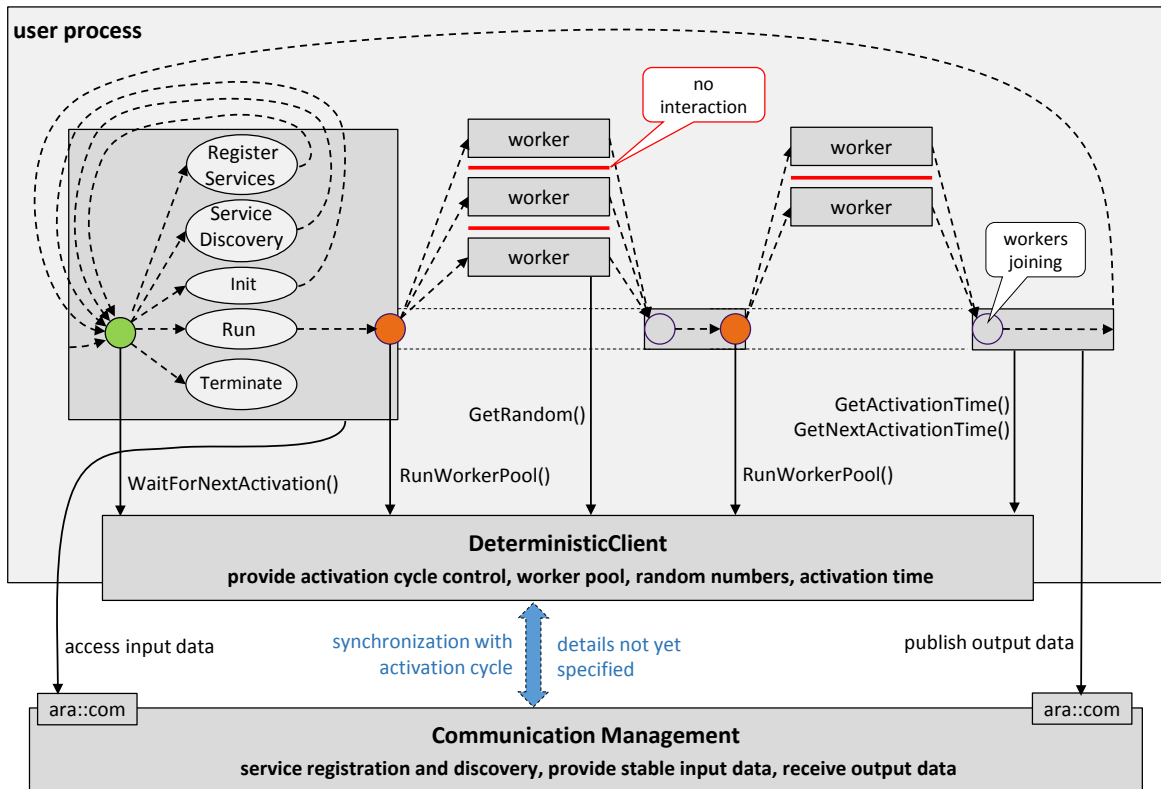


Figure 7.14: Cyclic Deterministic Execution

Cyclic *Process* behavior is controlled by a wait point API. The API returns a code to control the process mode (register services/ service discovery/ init/ run/ terminate). The execution is triggered by the *DeterministicClient*, depending on a defined period or on received events. Within a *Process*, all input data is available via *ara::com* (polling-based access only) when execution starts and is stable over one execution cycle. For details see 7.7.3.1.

The workload can be deployed to a worker pool API, which allows deterministic execution of a set of container elements (e.g. data sets), which are processed in parallel by the same runnable object (i.e. application function). The runnable object is not allowed to exchange any information while it is running, i.e. it doesn't access data which can be altered by other instances of the runnable object to avoid race conditions. The runnable object instances can physically run in parallel or sequentially in any order. For details see 7.7.3.2.

Additional *DeterministicClient* APIs provide random numbers and activation time stamps. Common HAD algorithms use particle filters which require random numbers. If used from within the worker pool, the random numbers are assigned to specific container elements to allow deterministic redundant execution. The activation time stamps don't change until the *Process* reaches its next wait point. For deterministic redundant execution, random number seeds and time stamps need to be synchronized. For details see 7.7.3.3 and 7.7.3.4.

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

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

The *DeterministicClient* Class is only local to the *Process*. Therefore, there is currently no security concern foreseen for this API.

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

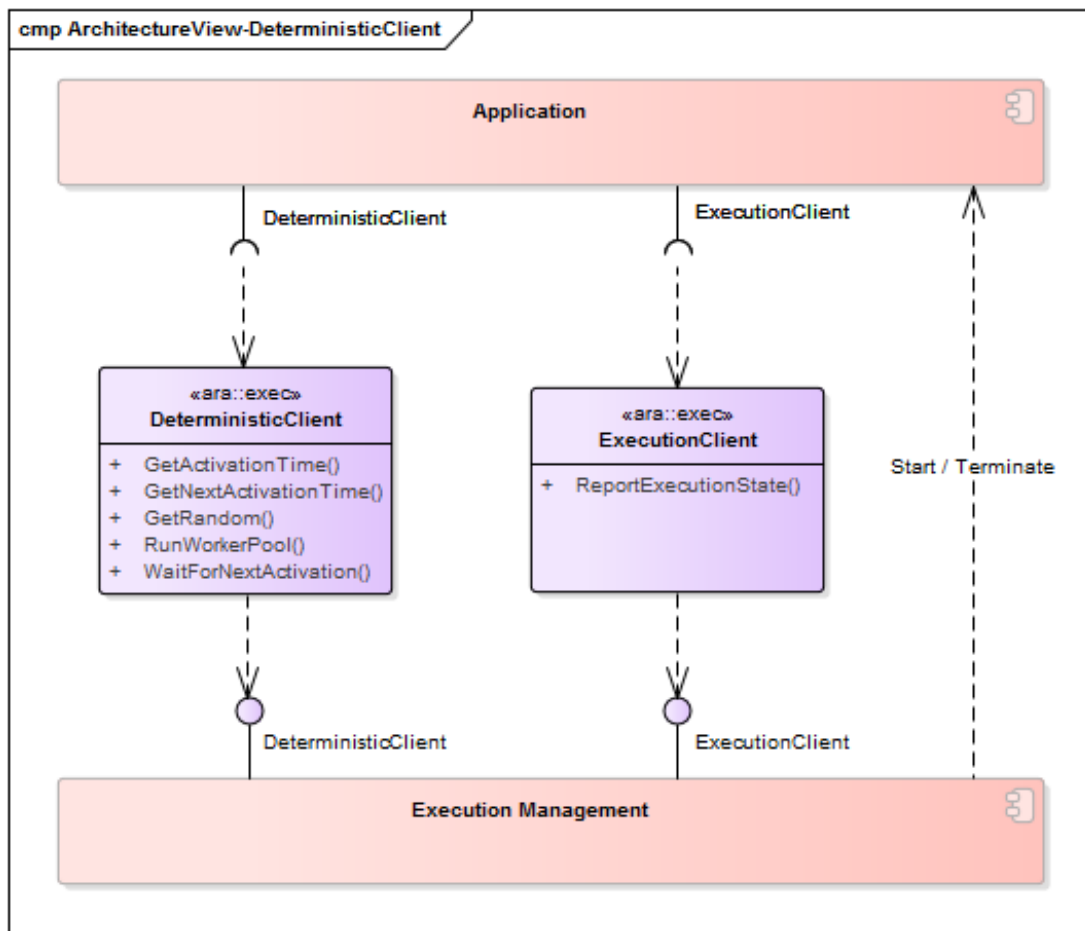


Figure 7.15: Deterministic Execution Interface

7.7.3 Cyclic Deterministic Execution

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

7.7.3.1 Control of Cyclic Execution

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

[SWS_EM_01301] Cyclic Execution [`Execution Management` shall provide a blocking wait point API `DeterministicClient::WaitForNextActivation`.] ([RS_EM_00052](#))

After the `Process` has been started by `Execution Management`, it reports `ExecutionState` `kRunning` (see [7.4.1](#)) and calls `DeterministicClient::WaitForNextActivation`.

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

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

[SWS_EM_01302] Cyclic Execution Control [`DeterministicClient::WaitForNextActivation` shall return a code to control the execution mode of the calling `Process`. Possible codes are `kRegisterServices`, `kServiceDiscovery`, `kInit`, `kRun`, and `kTerminate`.] ([RS_EM_00052](#))

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

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

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

Init – The `Process` initializes its internal data structures. The worker pool (see [7.7.3.2](#)) can be accessed once or several time sequentially. A budget (see [7.7.3.5](#)) needs to be assigned to the “Init” cycle.

Run – The `Process` performs one cycle of its normal cyclic execution. This can be repeated indefinitely. The worker pool (see [7.7.3.2](#)) can be accessed once or several times sequentially within a cycle. A budget (see [7.7.3.5](#)) needs to be assigned.

Terminate – The `Process` prepares to terminate. The actual termination is performed according to [SWS_EM_01404] or [SWS_EM_01405], see section 7.4.2.

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

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

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

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

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

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

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

[SWS_EM_01352]{DRAFT} Execution Cycle Timeout [If the `Process` calls `DeterministicClient::WaitForNextActivation` within a `kRun` cycle after the configured cycle time “`cycleTimeValue`” has been exceeded since the last activation, `Platform Health Management` shall be notified about the timeout to initiate appropriate recovery actions.]([RS_EM_00052](#))

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

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

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

7.7.3.2 Worker Pool

[SWS_EM_01305] Worker Pool [`Execution Management` shall provide a blocking API `DeterministicClient::RunWorkerPool` to run a deterministic worker pool to be used within the `Process` execution cycle.]([RS_EM_00053](#))

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

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

[SWS_EM_01306] Processing Container Objects
[`DeterministicClient::RunWorkerPool` shall sequentially (using the iterator of input parameter “`container`”) call a method `workerRunnable(...)` (input parameter

“runnableObj”) on every element of “container”, by using a worker pool of size “numberOfWorkers”.] ([RS_EM_00053](#))

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

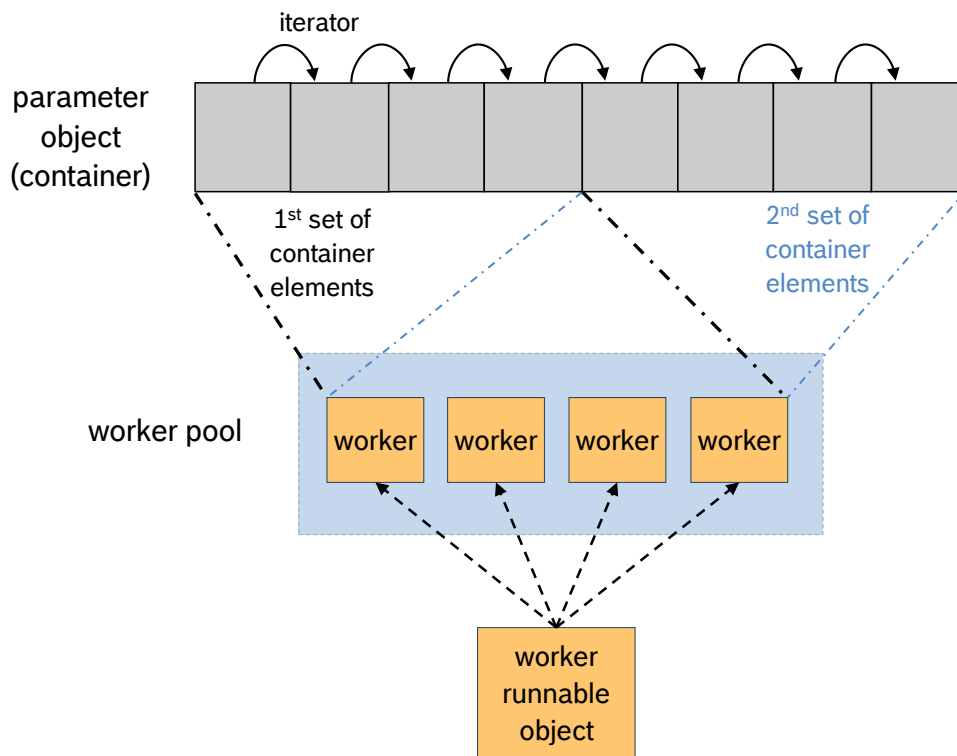


Figure 7.16: Worker Pool Usage

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

To achieve Data Determinism, the parallel workers need to satisfy certain implementation properties, e.g. no exchange of data is allowed between the instances of the runnable object which are processed by the workers. For details see [12]. Other, more complex solutions which allow interaction between the workers would be possible, but they increase complexity, reduce utilization and transparency, and are error-prone regarding the deterministic behavior.

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

An example for the implementation of a worker runnable object can be found in [12].

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

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

7.7.3.3 Random Numbers

[SWS_EM_01308] Random Numbers [Execution Management shall provide an API `DeterministicClient::GetRandom` which provides “Deterministic” random numbers. ‘Deterministic’ means, that the provided random numbers are identical for `Processes` which are executed redundantly, including within runnable objects being processed by a worker pool (see [SWS_EM_01305]).] (RS_EM_00053)

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

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

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

7.7.3.4 Time Stamps

The deterministic user `Process` might need timing information while cyclically (see 7.7.3.1) processing its input data in the `kRun` cycle. The used time value may have an influence on the calculated results. Therefore, `Execution Management` returns deterministic timestamps that represent the points in time when the current cycle was activated and when the next cycle will be activated, if this value is known. The timestamps are required to be identical for `Processes` which are executed redundantly, e.g. in a lockstep environment (see 7.7.2).

[SWS_EM_01310] Get Activation Time [Execution Management shall provide an API `DeterministicClient::GetActivationTime` which provides a deterministic timestamp that represents the point in time when the current `kRun` cycle was activated by `DeterministicClient::WaitForNextActivation` (see

[SWS_EM_01301]). Deterministic means, that the timestamps are identical for `Processes` which are executed redundantly. Subsequent calls within a cycle shall always return the same value.](RS_EM_00053)

[SWS_EM_01311] Activation Time Unknown [If `DeterministicClient::GetActivationTime` is called from outside a `kRun` cycle, `Execution Management` shall return `kNotAvailable`.](RS_EM_00053)

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

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

7.7.3.5 Real-Time Resources

To ensure Time Determinism (see 7.7.1.1), i.e. to make sure that a cyclic deterministic execution within a `Process` (see 7.7.3.1) is finished at a given deadline we need:

- `Execution Management` supports deterministic multithreading to meet high performance demand, see 7.7.3.2
- The integrator needs to assign appropriate resources to the `Process`.
- The integrator needs to assign appropriate scheduling policies. Details and options other than standard POSIX scheduling policies (see [SWS_EM_01014]) heavily depend on the used Operating System, are vendor specific, and are for now out of scope of the Adaptive Platform specification.
- The integrator needs to configure deadline monitoring, possibly execution budget monitoring, and appropriate recovery actions in case of violations. For more details on resources see 7.8.

To make sure that all `Processes` which use the `DeterministicClient` APIs get enough computing resources and can finish their cycle in time, it is in particular important to know when the worker pool (`DeterministicClient::RunWorkerPool`) is needed within a `kInit` and `kRun` `DeterministicClient::WaitForNextActivation` cycle. Also, a good computing resource utilization can only be achieved if usage of the workers (i.e. of available cores) can be distributed evenly over time. If the application code is known to the integrator, it should not be a problem to analyze the behavior and configure the system accordingly. However, if third party “black box” applications are delivered for

integration, their resource demands need to be described in a standardized way, so the integrator has a rough idea about the distribution of resource consumption within a `DeterministicClient::WaitForNextActivation-cycle`.

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

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

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

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

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

- `numberOfInstructions` [NormalizedInstructions]

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

- `speedup` = sequential runtime / parallelized runtime

Defines how much faster the calculations within one cycle can be finished if `numberOfWorkers` (see 7.7.3.2) are physically available, i.e. if enough cores were available on the machine to perform parallel execution of all workers.

- `sequentialInstructionsBegin` [NormalizedInstructions]

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

- `sequentialInstructionsEnd` [NormalizedInstructions]

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

Examples

Example 7.4

The `Process` uses the worker pool mainly in the middle of the cycle. The first 100 (normalized) instructions are mostly sequential, the next 275 instructions have a benefit when using the worker pool, and the last 125 instructions are mostly sequential again. The average speedup, over the complete 500 instructions is 1.3.

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

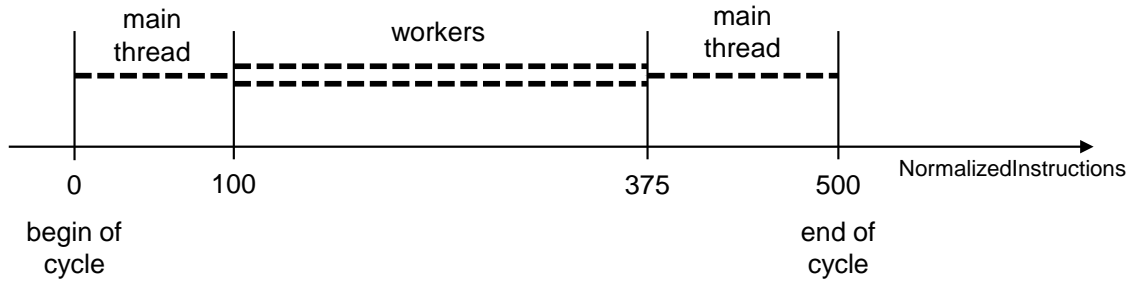


Figure 7.17: Worker pool used in middle of cycle

Example 7.5

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

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

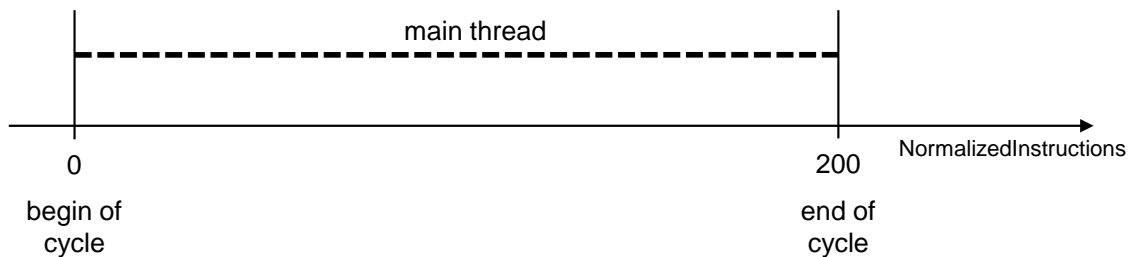


Figure 7.18: No benefit from worker pool

Example 7.6

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

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

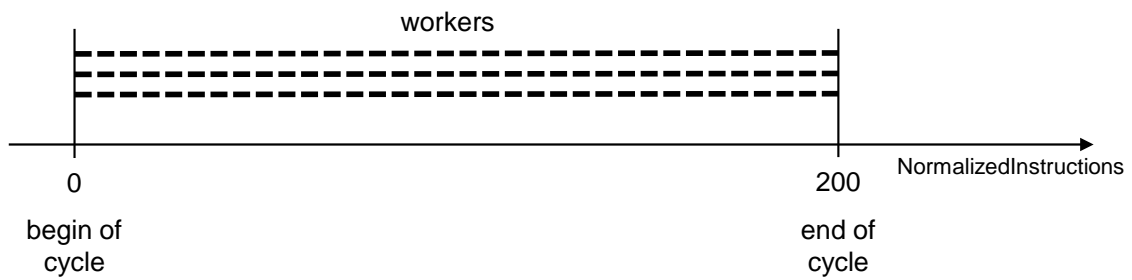


Figure 7.19: Full utilization of worker pool

7.8 Resource Limitation

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

However, [AUTOSAR Adaptive Platform](#) cannot support all mechanisms as described in this overview chapter in a standardized way, because the availability highly depends on the used Operating System.

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

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

7.8.1 Resource Configuration

This section provides an overview on resource assignment to [Processes](#). The resources considered in this specification are:

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

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

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

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

The following stakeholders are involved in resource management:

- Application Developer

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

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

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

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

- Integrator

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

- Assignment of resource criticality to [Processes](#), depending on safety and functional requirements.
- 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 [7.8.3.1](#).
- If the summarized maximum RAM needs of all [Processes](#), which can be running in parallel at any given time, exceeds the available RAM, this cannot be solved easily by prioritization, since memory assignment to low critical [Processes](#) cannot just be removed without compromising the [Process](#). However, it should be ensured that [Processes](#) with high criticality have ready access to their maximum resources at any time, while lower criticality [Processes](#) need to share the remaining resources. For details see [7.8.3.4](#).

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

is required to belong to a [ResourceGroup](#), clarifying how the [Application](#) will be constrained at the system level.

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

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

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

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

7.8.2 Resource Monitoring

As far as technically possible, the resources which are actually used by a [Process](#) should be controlled at any given time. For the entire system, the monitoring part of this activity is fulfilled by the Operating System. For details on CPU time monitoring see [7.8.3.1](#). For RAM monitoring see [7.8.3.4](#). The monitoring capabilities depend on the used Operating System. Depending on system requirements and safety goals, an appropriate Operating System has to be chosen and configured accordingly, in combination with other monitoring mechanisms (e.g. for execution deadlines) which are provided by Platform Health Management.

Resource monitoring can serve several purposes, e.g.

- Detection of misbehavior of the monitored [Process](#) to initiate appropriate recovery actions, like [Process](#) restart or state change, to maintain the provided functionality and guarantee functional safety.
- Protection of other parts of the system by isolating the erroneous [Processes](#) from unaffected ones to avoid resource shortage.

For [Processes](#) which are attempting to exceed their configured maximum resource needs (see [7.8.1](#)), one of the following alternatives is valid:

- The resource limit violation or deadline miss is considered a failure and recovery actions may need to be initiated. Therefore the specific violation gets reported to the Platform Health Management, which then starts recovery actions which have been configured beforehand. This will be the standard option for deterministic subsystems (see [7.7.1](#)).

- For [Processes](#) without hard deadlines, resource violations sometimes can be mitigated without dedicated error recovery actions, e.g. by interrupting execution and continue at a later point in time.
- If the OS provides a way to limit resource consumption of a [Process](#) or a group of [Processes](#) by design, explicit external monitoring is usually not necessary and often not even possible. Instead, the limitation mechanisms make sure that resource availability for other parts of the system is not affected by failures within the enclosed [Processes](#). When such by-design limitation is used, monitoring mechanisms may still be used for the benefit of the platform, but are not required. Self-monitoring and out-of-process monitoring is currently out-of-scope in [AUTOSAR Adaptive Platform](#).

7.8.3 Application-level Resource configuration

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

7.8.3.1 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](#).

7.8.3.2 Core Affinity

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

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

7.8.3.3 Scheduling

Currently available POSIX-compliant [Operating Systems](#) offer the scheduling policies required by POSIX, and in most cases additional, but different and incompatible scheduling strategies. This means for now, the required scheduling properties need to be configured individually, depending on the chosen OS.

Moreover, scheduling strategy is defined per thread and the POSIX standard allows for modifying the scheduling policy at runtime for a given thread, using `pthread_setschedparam()`. It is therefore not currently possible for the [AUTOSAR Adaptive Platform](#) to enforce a particular scheduling strategy for an entire process, but only for its first thread.

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

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

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

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

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

The available priority range and actual meaning of the scheduling priority depends on the selected scheduling policy, see [constr_1692], [TPS_MANI_01061] and [TPS_MANI_01188] in [4].

7.8.3.3.1 Resource Management

In general, for deterministic behavior the required computing time is guaranteed and violations are treated as error, while best-effort subsystems are more robust and might be able to mitigate sporadic violations, e.g. by continuing the calculation at the next activation, or by providing a result of lesser quality. This means, if time (e.g. deadline or runtime budget) monitoring is in place, the reaction on deviations is different for deterministic and best-effort subsystems.

In fact, it may not even be necessary to monitor best-effort subsystems, since they by definition are doing only a function that may not succeed. This leads to an architecture where monitoring is a voluntary, configured property.

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

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

[SWS_EM_02106]{DRAFT} ResourceGroup assignment [`Execution Management` shall configure the `Process` according to its `ResourceGroup` membership.] (`RS_EM_00005`)

7.8.3.4 Memory Budget and Monitoring

To render a function, a `Process` requires the availability of some amount of memory for its usage (mainly code, data, heap, thread stacks). Over the course of its execution however, not all of this memory is required at all times, such that an OS can take advantage of this property to make these ranges of memory available on-demand, and provide them to other `Processes` when the memory is no longer used.

While this has clear advantages in terms of system flexibility as well as memory efficiency, it is also in the way of both Time Determinism and Full Determinism: when a range of memory that was previously unused should now be made available, the OS may have to execute some amounts of potentially-unbounded activities to make this memory available. Often, the reverse may also be happening, removing previously available (but unused) memory from the `Process` under scope, to make it available to other `Processes`. This is detrimental to an overall system determinism.

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

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

In order to provide sufficient memory at the beginning of the execution of a `Process`, some properties may need to be defined for each `Process`.

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

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

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

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

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

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

7.8.3.5 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.

7.9 Fault Tolerance

7.9.1 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.

7.9.2 Scope

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

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

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

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

The AUTOSAR Adaptive Platform architecture is composed of two levels; Application and Platform Instance. The Application level constitutes cooperative Applications intended to satisfy 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.

7.9.3 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 until deployment. Other sources of faults include physical faults, e.g. random hardware failures, that

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

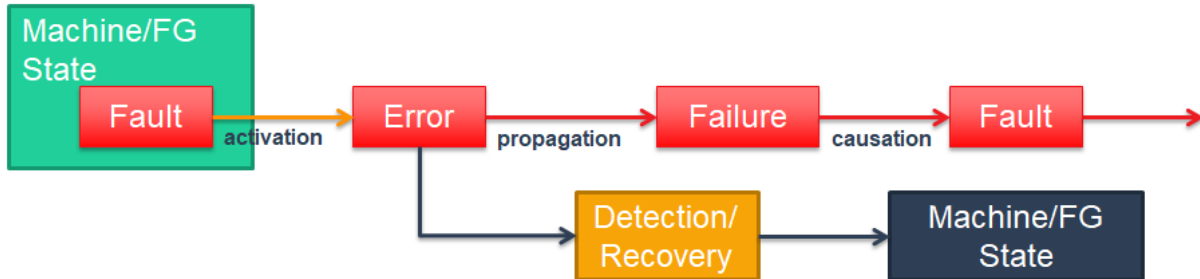


Figure 7.20: General Fault Tolerance scheme.

From the perspective of Execution Management, fault activation occurs 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.

7.10 Security

7.10.1 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 - shall 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]{DRAFT} Availability of a Trust Anchor [If there is no Trust Anchor available on the [Machine](#), the following requirements may be ignored: [\[SWS_EM_02300\]](#), [\[SWS_EM_02301\]](#), [\[SWS_EM_02302\]](#), [\[SWS_EM_02303\]](#), [\[SWS_EM_02304\]](#), [\[SWS_EM_02305\]](#), [\[SWS_EM_02306\]](#), [\[SWS_EM_02307\]](#), [\[SWS_EM_02308\]](#), [\[SWS_EM_02309\]](#).] ([RS_EM_00014](#))

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]{DRAFT} Integrity and Authenticity of processed [Machine Manifest](#) [[Execution Management](#) shall ensure that the integrity and authenticity of the processed [Machine Manifest](#) are checked.] ([RS_EM_00014](#))

[SWS_EM_02301]{DRAFT} Integrity and Authenticity of each [Executable](#) [[Execution Management](#) shall ensure that for every [Process](#) that is about to be started, the integrity and authenticity of the [Executable](#) itself are checked.] ([RS_EM_00014](#))

[SWS_EM_02302]{DRAFT} Integrity and Authenticity of shared objects [[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.] ([RS_EM_00014](#))

[SWS_EM_02303]{DRAFT} Integrity and Authenticity of processed [Execution Manifests](#) [[Execution Management](#) shall ensure that for every [Process](#) that is about to be started, the integrity and authenticity of its corresponding processed [Execution Manifests](#) are checked.] ([RS_EM_00014](#))

[SWS_EM_02304]{DRAFT} Integrity and Authenticity of processed Service Instance Manifests [Execution Management shall ensure that for every Process that is about to be started, the integrity and authenticity of its corresponding processed Service Instance Manifests are checked.] (RS_EM_00014)

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 contain Executable code that is executed within the context of the Process. 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 shall ensure that all Executable code on the Adaptive Platform is authenticated before being executed. The complete authenticated start-up sequence looks like this:

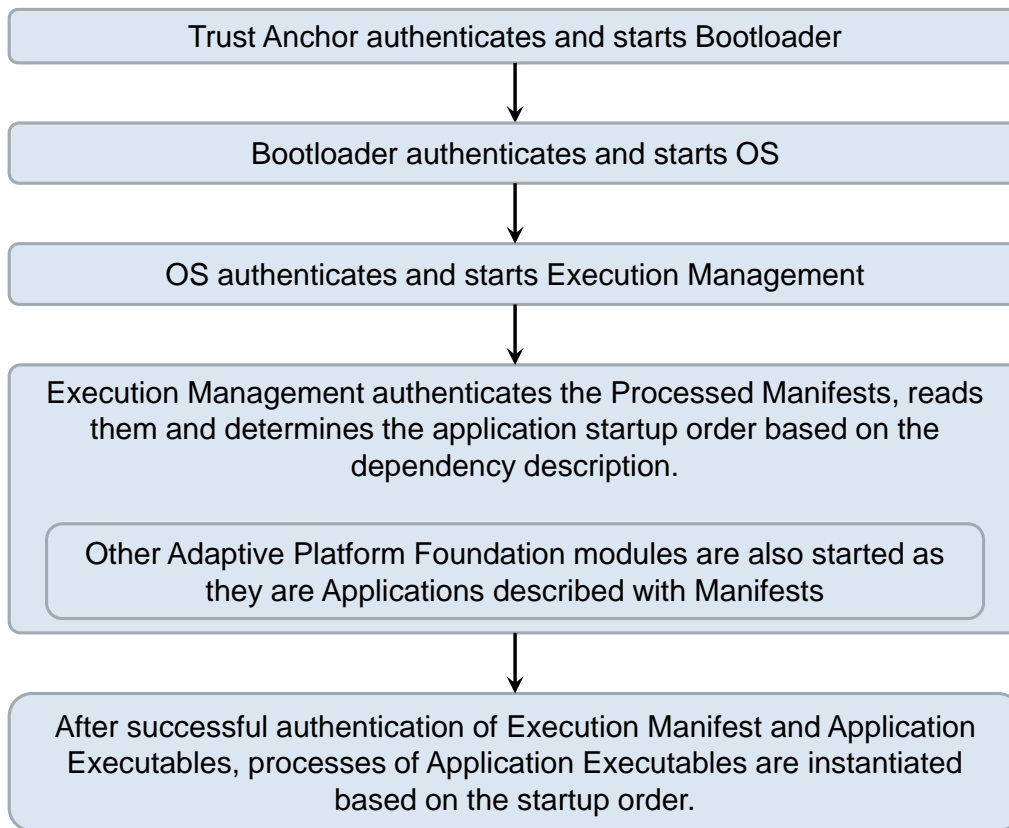


Figure 7.21: Authenticated start-up sequence

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.

7.10.1.1 Handling of failed authenticity checks

If the integrity and authenticity has been verified successfully, the system shall continue with its regular start-up process. If the integrity and authenticity check has failed, however, [Execution Management](#) shall offer a configuration option on how to proceed with the start-up process.

[SWS_EM_02305]{DRAFT} Failed authenticity checks [[Execution Management](#) shall offer two modes for handling failed authenticity checks: Monitoring Mode and Strict Mode.]([RS_EM_00014](#))

The configuration of the two modes is done via the [Machine Manifest](#). The configuration option shall only be processed after the integrity and authenticity of the [Machine Manifest](#) have been verified.

[SWS_EM_02306]{DRAFT} Machine Manifest [[Execution Management](#) shall stop the start-up sequence of the Adaptive Platform if the integrity or authenticity check of the processed [Machine Manifest](#) has failed.]([RS_EM_00014](#))

7.10.1.1.1 Monitoring Mode

In Monitoring Mode, 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.

Monitoring Mode 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.

Monitoring Mode 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.

7.10.1.1.2 Strict Mode

In Strict Mode, 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]{DRAFT} Strict Mode - Execution manifest [In Strict Mode, [Execution Management](#) shall not initiate the execution of an Executable if the integrity or authenticity check of the corresponding processed [Execution Manifest](#) has failed.] ([RS_EM_00014](#))

[SWS_EM_02308]{DRAFT} Strict Mode - Service Instance manifests [In Strict Mode, [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.] ([RS_EM_00014](#))

[SWS_EM_02309]{DRAFT} Strict Mode - Executables [In Strict Mode, [Execution Management](#) shall execute code only if its integrity and authenticity check was successful.] ([RS_EM_00014](#))

Executable code can be provided by executables, but also by shared objects that are linked by the executable.

Example: Consider an Adaptive Platform in Strict Mode. [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.

8 API specification

8.1 Type Definitions

8.1.1 ExecutionState

[SWS_EM_02000]{DRAFT} [

Kind:	enumeration	
Symbol:	ara::exec::ExecutionState	
Scope:	namespace ara::exec	
Underlying type:	uint8_t	
Syntax:	enum class ExecutionState : uint8_t {...};	
Values:	kRunning= 0	After a Process has been started by Execution Management, it reports ExecutionState kRunning.
	kTerminating= 1	On receipt of SIGTERM, a Reporting Process acknowledges the request (by reporting kTerminating to Execution Management.
Header file:	#include "ara/exec/execution_client.h"	
Description:	Defines the internal states of a Process (see 7.3.1). Scoped Enumeration of uint8_t .	

]([RS_EM_00103](#))

Please note that ExecutionState includes only states reportable by the [Process](#) to [Execution Management](#) and therefore does not include enumerations e.g. the "Initializing" state mentioned in figure 7.3 and 7.9, which are an implied states for [Execution Management](#). The Initializing state starts when [Process](#) is first scheduled (so no code executed yet) and ends when kRunning is reported ([[SWS_EM_01004](#)]). The Terminating state starts when [Process](#) first reports kTerminating or termination is requested by [Execution Management](#) and ends when the [Process](#) terminates ([[SWS_EM_01404](#)] and [[SWS_EM_01405](#)]). For the reasons mentioned, [Execution Management](#) assumes that [Process](#) is in initializing state until kRunning will be reported by it.

8.1.2 ActivationReturnType

[SWS_EM_02201]{DRAFT} [

Kind:	enumeration
Symbol:	ara::exec::ActivationReturnType
Scope:	namespace ara::exec
Underlying type:	uint8_t





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

](RS_EM_00052)

8.1.3 ActivationTimeStampReturntype

[SWS_EM_02202]{DRAFT} [

Kind:	enumeration	
Symbol:	ara::exec::ActivationTimeStampReturntype	
Scope:	namespace ara::exec	
Underlying type:	uint8_t	
Syntax:	enum class ActivationTimeStampReturntype : uint8_t {...};	
Values:	kSuccess= 0	–
	kNotAvailable= 1	–
Header file:	#include "ara/exec/deterministic_client.h"	
Description:	Defines the return codes for "get activation timestamp" operations. Scoped Enumeration of uint8_t .	

](RS_EM_00053)

8.2 Class Definitions

8.2.1 ExecutionClient class

The Execution State API provides the functionality for a [Process](#) to report its state to the [Execution Management](#).

[SWS_EM_02001]{DRAFT} [

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

](RS_EM_00103)

8.2.1.1 ExecutionClient::ExecutionClient

[SWS_EM_02030]{DRAFT} [

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

](RS_EM_00103)

8.2.1.2 ExecutionClient::~~ExecutionClient

[SWS_EM_02002]{DRAFT} [

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

](RS_EM_00103)

8.2.1.3 ExecutionClient::ReportExecutionState

[SWS_EM_02003]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::ExecutionClient::ReportExecutionState(ExecutionState state)	
Scope:	class ara::exec::ExecutionClient	
Syntax:	ara::core::Result<void> ReportExecutionState (ExecutionState state) const noexcept;	
Parameters (in):	state	Value of the Execution State
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:	noexcept	
Errors:	ara::exec::ExecErrc::kGeneralError	if some unspecified error occurred
Header file:	#include "ara/exec/execution_client.h"	
Description:	Interface for a Process to report its internal state to Execution Management.	

](RS_EM_00103)

8.2.2 DeterministicClient class

The `DeterministicClient` class provides the functionality for an `Application` to run a cyclic deterministic execution, see 7.7.3. Each `Process` which needs support for cyclic deterministic execution has to instantiate this class.

[SWS_EM_02210] [

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

](RS_EM_00052)

8.2.2.1 DeterministicClient::DeterministicClient

[SWS_EM_02211] [

Kind:	function
Symbol:	ara::exec::DeterministicClient::DeterministicClient()
Scope:	class ara::exec::DeterministicClient
Syntax:	DeterministicClient () const noexcept;
Exception Safety:	noexcept





Header file:	#include "ara/exec/deterministic_client.h"
Description:	Constructor for DeterministicClient which opens the Execution Management communication channel (e.g. POSIX FIFO) to access a wait point for cyclic execution, a worker pool, deterministic random numbers and time stamps .

](RS_EM_00052, RS_EM_00053)

8.2.2.2 DeterministicClient::~DeterministicClient

[SWS_EM_02215] [

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

](RS_EM_00052, RS_EM_00053)

8.2.2.3 DeterministicClient::WaitForNextActivation

[SWS_EM_02216] [

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

](RS_EM_00052)

8.2.2.4 DeterministicClient::RunWorkerPool

[SWS_EM_02220] [

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

]([RS_EM_00053](#))

8.2.2.5 DeterministicClient::GetRandom

[SWS_EM_02225] [

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

]([RS_EM_00053](#))

8.2.2.6 DeterministicClient::GetActivationTime

[SWS_EM_02230] [

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

](RS_EM_00053)

8.2.2.7 DeterministicClient::GetNextActivationTime

[SWS_EM_02235] [

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

](RS_EM_00053)

8.2.3 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, it's internal value stay bounded to it for entire lifetime of a object.

[SWS_EM_02263]{DRAFT} [

Kind:	class
Symbol:	ara::exec::FunctionGroup
Scope:	namespace ara::exec
Syntax:	<code>class FunctionGroup {...};</code>
Header file:	<code>#include "ara/exec/state_client.h"</code>
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.

|(RS_EM_00101)

8.2.3.1 FunctionGroup::Preconstruct

[SWS_EM_02264]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroup::Preconstruct(ara::core::StringView metaModelIdentifier)	
Scope:	class ara::exec::FunctionGroup	
Syntax:	<code>static Result<FunctionGroup::CtorToken> Preconstruct (ara::core::StringView metaModelIdentifier) noexcept;</code>	
Parameters (in):	metaModelIdentifier	stringified meta model identifier (short name path) where path separator is '/
Return value:	Result< FunctionGroup::CtorToken >	a construction token from which an instance of FunctionGroup can be constructed, or ExecErrc error.
Exception Safety:	noexcept	
Thread Safety:	Thread-safe	
Errors:	ara::exec::ExecErrc::kMetaModelError	if metaModelIdentifier passed is incorrect (e.g. FunctionGroupState identifier has been passed).
	ara::exec::ExecErrc::kGeneralError	if any other error occurs.
Header file:	<code>#include "ara/exec/state_client.h"</code>	
Description:	Pre construction method for FunctionGroup. This method shall validate/verify meta-model path passed and perform any operation that could fail and are expected to be performed in constructor.	

|(RS_EM_00101)

8.2.3.2 FunctionGroup::FunctionGroup

[SWS_EM_02265]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroup::FunctionGroup(FunctionGroup::CtorToken &&token)	
Scope:	class ara::exec::FunctionGroup	
Syntax:	FunctionGroup (FunctionGroup::CtorToken &&token) noexcept;	
Parameters (in):	token	representing pre-constructed object.
Exception Safety:	noexcept	
Header file:	#include "ara/exec/state_client.h"	
Description:	Constructor that creates FunctionGroup instance.	
Notes:	Please note that token is destructed during object construction!	

](RS_EM_00101)

8.2.3.3 FunctionGroup::~FunctionGroup

[SWS_EM_02266]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroup::~FunctionGroup()	
Scope:	class ara::exec::FunctionGroup	
Syntax:	~FunctionGroup () noexcept;	
Exception Safety:	noexcept	
Header file:	#include "ara/exec/state_client.h"	
Description:	Destructor of the FunctionGroup instance.	

](RS_EM_00101)

8.2.3.4 FunctionGroup::operator==

[SWS_EM_02267]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroup::operator==(FunctionGroup const &other)	
Scope:	class ara::exec::FunctionGroup	
Syntax:	bool operator== (FunctionGroup const &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:	noexcept	
Thread Safety:	Thread-safe	
Header file:	#include "ara/exec/state_client.h"	





Description:	eq operator to compare with other FunctionGroup instance.
---------------------	---

](RS_EM_00101)

8.2.3.5 FunctionGroup::operator!=

[SWS_EM_02268]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroup::operator!=(FunctionGroup const &other)	
Scope:	class ara::exec::FunctionGroup	
Syntax:	bool operator!=(FunctionGroup const &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:	noexcept	
Thread Safety:	Thread-safe	
Header file:	#include "ara/exec/state_client.h"	
Description:	uneq operator to compare with other FunctionGroup instance.	

](RS_EM_00101)

8.2.4 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, it's internal value stay bounded to it for entire lifetime of a object.

[SWS_EM_02269]{DRAFT} [

Kind:	class
Symbol:	ara::exec::FunctionGroupState
Scope:	namespace ara::exec
Syntax:	class FunctionGroupState {...};
Header file:	#include "ara/exec/state_client.h"
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.

](RS_EM_00101)

8.2.4.1 FunctionGroupState::Preconstruct

[SWS_EM_02270]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroupState::Preconstruct(FunctionGroup const &functionGroup, ara::core::StringView metaModelIdentifier)	
Scope:	class ara::exec::FunctionGroupState	
Syntax:	static ara::core::Result<FunctionGroupState::CtorToken> Preconstruct (FunctionGroup const &functionGroup, ara::core::StringView metaModelIdentifier) noexcept;	
Parameters (in):	functionGroup	the Function Group instance the state shall be connected with.
	metaModelIdentifier	stringified meta model identifier (short name path) where path separator is '/'.
Return value:	ara::core::Result< FunctionGroup State::CtorToken >	a construction token from which an instance of FunctionGroupState can be constructed, or Exec ErrorDomain error.
Exception Safety:	noexcept	
Thread Safety:	Thread-safe	
Errors:	ara::exec::ExecErrc::kMetaModelError	if metaModelIdentifier passed is incorrect (e.g. FunctionGroup identifier has been passed).
	ara::exec::ExecErrc::kGeneralError	if any other error occurs.
Header file:	#include "ara/exec/state_client.h"	
Description:	Pre construction method for FunctionGroupState. This method shall validate/verify meta-model path passed and perform any operation that could fail and are expected to be performed in constructor.	

](RS_EM_00101)

8.2.4.2 FunctionGroupState::FunctionGroupState

[SWS_EM_02271]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroupState::FunctionGroupState(FunctionGroupState::CtorToken &&token)	
Scope:	class ara::exec::FunctionGroupState	
Syntax:	FunctionGroupState (FunctionGroupState::CtorToken &&token) noexcept;	
Parameters (in):	token	representing pre-constructed object.
Exception Safety:	noexcept	
Header file:	#include "ara/exec/state_client.h"	
Description:	Constructor that creates FunctionGroupState instance.	
Notes:	Please note that token is destructed during object construction!	

](RS_EM_00101)

8.2.4.3 FunctionGroupState::~~FunctionGroupState

[SWS_EM_02272]{DRAFT} [

Kind:	function
Symbol:	ara::exec::FunctionGroupState::~~FunctionGroupState()
Scope:	class ara::exec::FunctionGroupState
Syntax:	~FunctionGroupState () noexcept;
Exception Safety:	noexcept
Header file:	#include "ara/exec/state_client.h"
Description:	Destructor of the FunctionGroup instance.

]([RS_EM_00101](#))

8.2.4.4 FunctionGroupState::operator==

[SWS_EM_02273]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroupState::operator==(FunctionGroupState const &other)	
Scope:	class ara::exec::FunctionGroupState	
Syntax:	bool operator== (FunctionGroupState const &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:	noexcept	
Thread Safety:	Thread-safe	
Header file:	#include "ara/exec/state_client.h"	
Description:	eq operator to compare with other FunctionGroupState instance.	

]([RS_EM_00101](#))

8.2.4.5 FunctionGroupState::operator!=

[SWS_EM_02274]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::FunctionGroupState::operator!=(FunctionGroupState const &other)	
Scope:	class ara::exec::FunctionGroupState	
Syntax:	bool operator!= (FunctionGroupState const &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:	noexcept	
Thread Safety:	Thread-safe	
Header file:	#include "ara/exec/state_client.h"	
Description:	uneq operator to compare with other FunctionGroupState instance.	

](RS_EM_00101)

8.2.5 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]{DRAFT} [

Kind:	class
Symbol:	ara::exec::StateClient
Scope:	namespace ara::exec
Syntax:	class StateClient {...};
Header file:	#include "ara/exec/state_client.h"
Description:	Class representing connection to Execution Management that is used to request Function Group state transitions (or other operations).
Notes:	StateClient opens communication channel to Execution Management (e.g. POSIX FIFO). Each Process that intends to perform state management, shall create an instance of this class and it shall have rights to use it.

](RS_EM_00101)

8.2.5.1 StateClient::StateClient

[SWS_EM_02276]{DRAFT} [

Kind:	function
Symbol:	ara::exec::StateClient::StateClient()
Scope:	class ara::exec::StateClient
Syntax:	StateClient () noexcept;
Exception Safety:	noexcept
Header file:	#include "ara/exec/state_client.h"
Description:	Constructor that creates State Client instance.

](RS_EM_00101)

8.2.5.2 StateClient::~~StateClient

[SWS_EM_02277]{DRAFT} [

Kind:	function
Symbol:	ara::exec::StateClient::~~StateClient()
Scope:	class ara::exec::StateClient
Syntax:	~StateClient () noexcept;
Exception Safety:	noexcept
Header file:	#include "ara/exec/state_client.h"
Description:	Destructor of the State Client instance.

](RS_EM_00101)

8.2.5.3 StateClient::SetState

[SWS_EM_02278]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::StateClient::SetState(FunctionGroupState const &state)	
Scope:	class ara::exec::StateClient	
Syntax:	ara::core::Future<void> SetState (FunctionGroupState const &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 ExecErrorDomain error.
Exception Safety:	noexcept	
Thread Safety:	Thread-safe	
Errors:	ara::exec::ExecErrc::kCancelled	if transition to the requested Function Group state was cancelled by a newer request





	<code>ara::exec::ExecErrc::kFailed</code>	if transition to the requested Function Group state failed
	<code>ara::exec::ExecErrc::kInvalidArguments</code>	if arguments passed doesn't appear to be valid (e.g. after a software update, given <code>functionGroup</code> doesn't exist anymore)
	<code>ara::exec::ExecErrc::kCommunicationError</code>	if <code>StateClient</code> can't communicate with Execution Management (e.g. IPC link is down)
	<code>ara::exec::ExecErrc::kGeneralError</code>	if any other error occurs.
Header file:	#include "ara/exec/state_client.h"	
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 <code>ara::core::Future</code> can be used to determine result of requested transition.	

]([RS_EM_00101](#))

Asynchronous nature of `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. Once finished it is implementation specific if answer will arrive on the corresponding future.

Since [Execution Management](#) allows to change direction of the ongoing [Function Group](#) state transition, it may happen (especially in misconfigured system, or during the development phase) that some of `StateClient::SetState` requests will be issued by mistake. It is in the best interest of [Execution Management](#) to inform requester (instance of `StateClient`) of the ongoing transition, that it had been canceled by a newer request as soon as possible.

[SWS_EM_02298]{DRAFT} Canceling ongoing state transition [When [Execution Management](#) receives `StateClient::SetState` request for a [Function Group](#) that is already under state transition. [Execution Management](#) shall cancel the ongoing state transition, by sending `kCancelled` transition result to requestor, before accepting new request.]([RS_EM_00101](#))

Please note that [\[SWS_EM_02298\]](#) merely ensures that [Execution Management](#) first informs requester of the ongoing transition (instance of `StateClient`) about cancellation, before informing new requester that the new request has been accepted. Both requesters could be the same instance of `StateClient`. There are no other requirements or assumptions on order in which requests from `StateClient::SetState` are processed.

8.2.5.4 `StateClient::GetInitialMachineStateTransitionResult`

[SWS_EM_02279]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::StateClient::GetInitialMachineStateTransitionResult()	
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 ExecErrorDomain error.
Exception Safety:	noexcept	
Thread Safety:	Thread-safe	
Errors:	ara::exec::ExecErrc::kCancelled	if transition to the requested Function Group state was cancelled by a newer request
	ara::exec::ExecErrc::kFailed	if transition to the requested Function Group state failed
	ara::exec::ExecErrc::kCommunication Error	if StateClient can't communicate with Execution Management (e.g. IPC link is down)
	ara::exec::ExecErrc::kGeneralError	if any other error occurs.
Header file:	#include "ara/exec/state_client.h"	
Description:	Method to retrieve result of Machine State initial transition to Startup state. This method allows State Management to retrieve 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 result delivered by this method shall not change (unless machine is started again).	

]([RS_EM_00101](#))

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

8.3 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 [8.3.1](#).

8.3.1 Execution Management error codes

[SWS_EM_02281]{DRAFT} [

Kind:	enumeration	
Symbol:	<code>ara::exec::ExecErrc</code>	
Scope:	namespace <code>ara::exec</code>	
Underlying type:	<code>ara::core::ErrorDomain::CodeType</code>	
Syntax:	<code>enum class ExecErrc : ara::core::ErrorDomain::CodeType { ...};</code>	
Values:	<code>kGeneralError= 1</code>	Some unspecified error occurred
	<code>kInvalidArguments= 2</code>	Invalid argument was passed
	<code>kCommunicationError= 3</code>	Communication error occurred
	<code>kMetaModelError= 4</code>	Wrong meta model identifier passed to a function
	<code>kCancelled= 5</code>	Transition to the requested Function Group state was cancelled by a newer request
	<code>kFailed= 6</code>	Transition to the requested Function Group state failed
Header file:	<code>#include "ara/exec/exec_error_domain.h"</code>	
Description:	Defines an enumeration class for the Execution Management error codes.	

]([RS_AP_00130](#), [RS_AP_00122](#), [RS_AP_00127](#))

8.3.2 ExecException type

[SWS_EM_02282]{DRAFT} [

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

]([RS_AP_00130](#), [RS_AP_00122](#), [RS_AP_00127](#))

8.3.2.1 ExecException::ExecException

[SWS_EM_02283]{DRAFT} [

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

]([RS_AP_00120](#), [RS_AP_00121](#), [RS_AP_00130](#), [RS_AP_00132](#))

8.3.3 GetExecErrorDomain function

[SWS_EM_02290]{DRAFT} [

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

]([RS_AP_00120](#), [RS_AP_00130](#), [RS_AP_00132](#))

8.3.4 MakeErrorCode function

[SWS_EM_02291]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::MakeErrorCode(ara::exec::ExecErrc code, ara::core::ErrorDomain::SupportDataType data)	
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:	noexcept	
Header file:	#include "ara/exec/exec_error_domain.h"	
Description:	Creates an instance of ErrorCode.	

](RS_AP_00120, RS_AP_00121, RS_AP_00130, RS_AP_00132)

8.3.5 ExecErrorDomain type

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

[SWS_EM_02284]{DRAFT} [

Kind:	class
Symbol:	ara::exec::ExecErrorDomain
Scope:	namespace ara::exec
Base class:	ara::core::ErrorDomain
Syntax:	<code>class ExecErrorDomain final : public ErrorDomain {...};</code>
Unique ID:	0x8000'0000'0000'0300ULL
Header file:	#include "ara/exec/exec_error_domain.h"
Description:	Defines a class representing the Execution Management error domain.

](RS_AP_00130, RS_AP_00122, RS_AP_00127)

8.3.5.1 ExecErrorDomain::ExecErrorDomain

[SWS_EM_02286]{DRAFT} [

Kind:	function
Symbol:	ara::exec::ExecErrorDomain::ExecErrorDomain()
Scope:	class ara::exec::ExecErrorDomain
Syntax:	<code>ExecErrorDomain () noexcept;</code>
Exception Safety:	noexcept
Header file:	#include "ara/exec/exec_error_domain.h"
Description:	Constructs a new ExecErrorDomain object.

](RS_AP_00120, RS_AP_00130, RS_AP_00132)

8.3.5.2 ExecErrorDomain::Name

[SWS_EM_02287]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::ExecErrorDomain::Name()	
Scope:	class ara::exec::ExecErrorDomain	
Syntax:	char const* Name () const noexcept override;	
Return value:	char const *	The name of the error domain.
Exception Safety:	noexcept	
Header file:	#include "ara/exec/exec_error_domain.h"	
Description:	Returns a string constant associated with ExecErrorDomain.	

]([RS_AP_00120](#), [RS_AP_00130](#), [RS_AP_00132](#))

[SWS_EM_02292]{DRAFT} [ExecErrorDomain::Name shall return the NULL-terminated string "Exec".]([RS_AP_00128](#))

8.3.5.3 ExecErrorDomain::Message

[SWS_EM_02288]{DRAFT} [

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

]([RS_AP_00120](#), [RS_AP_00121](#), [RS_AP_00130](#), [RS_AP_00132](#))

8.3.5.4 ExecErrorDomain::ThrowAsException

[SWS_EM_02289]{DRAFT} [

Kind:	function	
Symbol:	ara::exec::ExecErrorDomain::ThrowAsException(ara::core::ErrorCode const &errorCode)	
Scope:	class ara::exec::ExecErrorDomain	



△

Syntax:	void ThrowAsException (ara::core::ErrorCode const &errorCode) const noexcept(false) override;	
Parameters (in):	errorCode	The error to throw.
Return value:	None	
Exception Safety:	noexcept(false)	
Header file:	#include "ara/exec/exec_error_domain.h"	
Description:	Creates a new instance of ExecException from errorCode and throws it as a C++ exception.	

|(RS_AP_00120, RS_AP_00121, RS_AP_00130)

9 Service Interfaces

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

There are no service interfaces defined in this release.

A Mentioned Manifest Elements

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

Enumeration	CommandLineOptionKindEnum
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest
Note	This enum defines the different styles how the command line option appears in the command line. Tags: atp.Status=draft
Literal	Description
commandLineLong Form	Long form of command line option. Example: <nowiki>--version=1.0</nowiki> <ul style="list-style-type: none"> • <code>- help</nowiki></code> Tags: atp.EnumerationLiteralIndex=1
commandLineShort Form	Short form of command line option. Example: <nowiki>-v 1.0</nowiki> <ul style="list-style-type: none"> • <code>h</nowiki></code> Tags: atp.EnumerationLiteralIndex=0
commandLine SimpleForm	In this case the command line option does not have any formal structure. Just the value is passed to the program. Tags: atp.EnumerationLiteralIndex=2

Table A.1: CommandLineOptionKindEnum

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

Table A.2: DeterministicClient

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

Table A.3: DeterministicClientResource

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

Table A.4: DeterministicClientResourceNeeds

Class	Executable
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure





Class	Executable			
Note	This meta-class represents an executable program. Tags: atp.Status=draft atp.recommendedPackage=Executables			
Base	ARElement, ARObject, AtpClassifier, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Referrable			
Attribute	Type	Mult.	Kind	Note
buildType	BuildTypeEnum	0..1	attr	This attribute describes the buildType of a module and/or platform implementation.
loggingBehavior	LoggingBehaviorEnum	0..1	attr	This attribute indicates the intended logging behavior of 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. Tags: atp.Status=draft
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. Tags: atp.Status=draft
version	StrongRevisionLabel String	0..1	attr	Version of the executable. Tags: atp.Status=draft

Table A.5: Executable

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 Tags: atp.Status=draft
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.6: ExecutionStateReportingBehaviorEnum

Class	Machine
Package	M2::AUTOSARTemplates::AdaptivePlatform::MachineManifest





Class		Machine		
Note	Machine that represents an Adaptive Autosar Software Stack. Tags: atp.ManifestKind=MachineManifest atp.Status=draft atp.recommendedPackage=Machines			
Base	<i>ARElement, ARObject, AtpClassifier, AtpFeature, AtpStructureElement, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Referrable</i>			
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. Tags: atp.Status=draft
environment Variable	TagWithOptionalValue	*	aggr	This aggregation represents the collection of environment variables that shall be added to the environment defined on the level of the enclosing Machine. Stereotypes: atpSplitable Tags: atp.Splitkey=environmentVariable atp.Status=draft
functionGroup	ModeDeclarationGroup Prototype	*	aggr	This aggregation represents the collection of function groups of the enclosing Machine. Stereotypes: atpSplitable; atpVariation Tags: atp.Splitkey=shortName, variationPoint.shortLabel atp.Status=draft vh.latestBindingTime=preCompileTime
hwElement	HwElement	*	ref	This reference is used to describe the hardware resources of the machine. Stereotypes: atpUriDef Tags: atp.Status=draft
machineDesign	MachineDesign	1	ref	Reference to the MachineDesign this Machine is implementing. Tags: atp.Status=draft
module Instantiation	AdaptiveModule Instantiation	*	aggr	Configuration of Adaptive Autosar module instances that are running on the machine. Stereotypes: atpSplitable Tags: atp.Splitkey=shortName atp.Status=draft
processor	Processor	1..*	aggr	This represents the collection of processors owned by the enclosing machine. Tags: atp.Status=draft
secure Communication Deployment	SecureCommunication Deployment	*	aggr	Deployment of secure communication protocol configuration settings to crypto module entities. Stereotypes: atpSplitable Tags: atp.Splitkey=shortName atp.Status=draft





Class	Machine			
trustedPlatform Executable LaunchBehavior	TrustedPlatform ExecutableLaunch BehaviorEnum	1	attr	This attribute controls the behavior of how authentication affects the ability to launch for each Executable.

Table A.7: Machine

Class	ModeDeclaration			
Package	M2::AUTOSARTemplates::CommonStructure::ModeDeclaration			
Note	Declaration of one Mode. The name and semantics of a specific mode is not defined in the meta-model. Tags: atp.ManifestKind=ExecutionManifest,MachineManifest			
Base	ARObject, AtpClassifier, AtpFeature, AtpStructureElement, Identifiable, MultilanguageReferrable, Referrable			
Attribute	Type	Mult.	Kind	Note
–	–	–	–	–

Table A.8: ModeDeclaration

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

Table A.9: NormalizedInstruction

Class	Process			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class provides information required to execute the referenced executable. Tags: atp.ManifestKind=ExecutionManifest atp.Status=draft atp.recommendedPackage=Processes			
Base	ARElement, ARObject, AbstractExecutionContext, AtpClassifier, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Referrable, UploadablePackageElement			
Attribute	Type	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. Tags: atp.Status=draft





Class	Process			
deterministic Client	DeterministicClient	0..1	ref	This reference adds further execution characteristics for deterministic clients. Tags: atp.Status=draft
executable	Executable	0..1	ref	Reference to executable that is executed in the process. Stereotypes: atpUriDef Tags: atp.Status=draft
logTraceDefault LogLevel	LogTraceDefaultLog LevelEnum	0..1	attr	This attribute allows to set the initial log reporting level for a logTraceProcessId (ApplicationId).
logTraceFile Path	UriString	0..1	attr	This attribute defines the destination file to which the logging information is passed.
logTraceLog Mode	LogTraceLogMode Enum	*	attr	This attribute defines the destination of log messages provided by the process.
logTrace ProcessDesc	String	0..1	attr	This attribute can be used to describe the logTrace ProcessId that is used in the log and trace message in more detail.
logTrace ProcessId	String	0..1	attr	This attribute identifies the process in the log and trace message (ApplicationId).
numberOf RestartAttempts	PositiveInteger	0..1	attr	This attribute defines how often a process shall be restarted if the start fails. <nowiki>numberOfRestartAttempts = "0" OR Attribute not existing, start once numberOfRestartAttempts = "1", start a second time </nowiki>
preMapping	Boolean	0..1	attr	This attribute describes whether the executable is preloaded into the memory.
processState Machine	ModeDeclarationGroup Prototype	0..1	aggr	Set of Process States that are defined for the process. Tags: atp.Status=draft
stateDependent StartupConfig	StateDependentStartup Config	*	aggr	Applicable startup configurations. Tags: atp.Status=draft

Table A.10: Process

Class	StartupConfig			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class represents a reusable startup configuration for processes.. Tags: atp.ManifestKind=ExecutionManifest atp.Status=draft			
Base	<i>ARObject, Identifiable, MultilanguageReferrable, Referrable</i>			
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. Tags: atp.Status=draft
scheduling Policy	SchedulingPolicyKind Enum	0..1	attr	This attribute represents the ability to define the scheduling policy for the initial thread of the application.





Class	StartupConfig			
scheduling Priority	Integer	0..1	attr	This is the scheduling priority requested by the application itself.
startupOption	StartupOption	*	aggr	Applicable startup options Tags: atp.Status=draft
timeout	EnterExitTimeout	0..1	aggr	This aggregation can be used to specify the timeouts for launching and terminating the process depending on the StartupConfig. Tags: atp.Status=draft

Table A.11: StartupConfig

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

Table A.12: StartupOption

Class	StateDependentStartupConfig			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class defines the startup configuration for the process depending on a collection of machine states. Tags: atp.ManifestKind=ExecutionManifest atp.Status=draft			
Base	ARObject			
Attribute	Type	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. Tags: atp.Status=draft
functionGroup State	ModeDeclaration	*	iref	This represent the applicable functionGroupMode. Tags: atp.Status=draft





Class	StateDependentStartupConfig			
resourceConsumption	ResourceConsumption	0..1	aggr	This aggregation provides the ability to define resource consumption boundaries on a per-process-startup-config basis. Tags: atp.Status=draft
resourceGroup	ResourceGroup	1	ref	Reference to an applicable resource group. Tags: atp.Status=draft
startupConfig	StartupConfig	1	ref	Reference to a reusable startup configuration with startup parameters. Tags: atp.Status=draft

Table A.13: StateDependentStartupConfig

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

Table A.14: TagWithOptionalValue

B Interfaces to other Functional Clusters (informative)

B.1 Overview

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

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

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



Figure B.1: Interfaces between Functional Clusters

B.2 Interface Tables

B.2.1 Get Process States Information

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

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

Table B.1: Get Process States Information

B.2.2 Enter Unrecoverable State Request

	Name	Description	Requirements
Intended users	Platform Health Management	This is a "Recovery Action"	
Name proposal	*EnterUnrecoverableState*		
Functionality	Requests a change to a Unrecoverable State and stops all currently running processes		[SWS_EM_02261] [SWS_EM_02262]
Parameters (in)	None		
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation failed	This API will never return in normal operation. Therefore any actual return always indicate an error.	

Table B.2: Enter Unrecoverable State Request

B.2.3 Process Restart Request

	Name	Description	Requirements
Intended users	Platform Health Management	This is a "Recovery Action"	
Name proposal	*ProcessRestart*		

Functionality	Request to restart a Process	Restart a specific Process on the request from the Platform Health Management.	[SWS_EM_01016] [SWS_EM_01062]
Parameters (in)	Process identifier	Unique named identifier of the Process to be restarted. Not the PID because this will change.	[SWS_EM_01016]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_01064]
	general error	Process could not be restarted	[SWS_EM_01063]

Table B.3: Process Restart Request

C History of Constraints and Specification Items

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

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

C.1.1 Added Traceables in 17-10

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





Number	Heading
[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.1: Added Traceables in 17-10

C.1.2 Changed Traceables in 17-10

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



△

Number	Heading
[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.2: Changed Traceables in 17-10

C.1.3 Deleted Traceables in 17-10

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

Table C.3: Deleted Traceables in 17-10

C.1.4 Added Constraints in 17-10

none

C.1.5 Changed Constraints in 17-10

none

C.1.6 Deleted Constraints in 17-10

none

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

C.2.1 Added Traceables in 18-03

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





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





Number	Heading
[SWS_EM_02235]	DeterministicClient::GetNextActivationTime API

Table C.4: Added Traceables in 18-03

C.2.2 Changed Traceables in 18-03

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





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

Table C.5: Changed Traceables in 18-03

C.2.3 Deleted Traceables in 18-03

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



△

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

Table C.6: Deleted Traceables in 18-03

C.2.4 Added Constraints in 18-03

none

C.2.5 Changed Constraints in 18-03

none

C.2.6 Deleted Constraints in 18-03

none

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

C.3.1 Added Traceables in 18-10

none

C.3.2 Changed Traceables in 18-10

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





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

Table C.7: Changed Traceables in 18-10

C.3.3 Deleted Traceables in 18-10

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

Table C.8: Deleted Traceables in 18-10

C.3.4 Added Constraints in 18-10

none

C.3.5 Changed Constraints in 18-10

none

C.3.6 Deleted Constraints in 18-10

none

C.4 Constraint and Specification Item History of this document according to AUTOSAR Release 19-03

C.4.1 Added Traceables 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.9: Added Traceables in R19-03

C.4.2 Changed Traceables 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
[SWS_EM_01110]	Off States
[SWS_EM_01400]	Execution Dependency resolution
[SWS_EM_02000]	
[SWS_EM_02001]	
[SWS_EM_02201]	
[SWS_EM_02202]	





Number	Heading
[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.10: Changed Traceables in R19-03

C.4.3 Deleted Traceables 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.11: Deleted Traceables in R19-03

C.4.4 Added Constraints in R19-03

none

C.4.5 Changed Constraints in R19-03

none

C.4.6 Deleted Constraints in R19-03

none

C.5 Constraint and Specification Item History of this document according to AUTOSAR Release 19-11

C.5.1 Added Traceables 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]	
[SWS_EM_02275]	





Number	Heading
[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.12: Added Traceables in R19-11

C.5.2 Changed Traceables in R19-11

Number	Heading
[SWS_EM_01000]	Startup order
[SWS_EM_01001]	Execution Dependency error
[SWS_EM_01002]	Idle Process State





Number	Heading
[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
[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





Number	Heading
[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
[SWS_EM_02254]	Misconfigured Process - assigned to more than one Function Group





Number	Heading
[SWS_EM_02255]	State transition - Process termination timeout reaction

Table C.13: Changed Traceables in R19-11

C.5.3 Deleted Traceables 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.14: Deleted Traceables in R19-11

C.5.4 Added Constraints in R19-11

none

C.5.5 Changed Constraints in R19-11

none

C.5.6 Deleted Constraints in R19-11

none