| **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** | R20-11 |

**Document Change History**

| Date | Release | Changed by | Description |
|---|---|---|---|
| 2020-11-30 | R20-11 | AUTOSAR Release Management | • Further refinement of State Management API and semantics<br>• Update process lifecycle (terminating report optional)<br>• Added Deterministic Synchronization support<br>• EM-PHM interaction |
| 2019-11-28 | R19-11 | AUTOSAR Release | • Further refinement of State Management API and semantics<br>• Introduced support for trusted platform<br>• Added support for non-reporting Processes<br>• Execution Management API uses Core types<br>• Changed Document Status from Final to published |
| 2019-03-29 | 19-03 | AUTOSAR Release Management | • Refinement of State Management semantics<br>• Document structure modified to reflect current template |

| | | | |
|---|---|---|---|
| 2018-10-31 | 18-10 | AUTOSAR Release Management | • Refinement of Deterministic Execution<br>• Updated Process lifecycle to clarify Process and Execution States<br>• Updated Application Recovery Actions |
| 2018-03-29 | 18-03 | AUTOSAR Release Management | • Deterministic Execution<br>• Resource Limitation<br>• State Management<br>• Fault Tolerance elaboration |
| 2017-10-27 | 17-10 | AUTOSAR Release Management | • State Management elaboration, introduction of Function Groups<br>• Recovery actions for Platform Health Management<br>• Resource limitation and deterministic execution |
| 2017-03-31 | 17-03 | AUTOSAR Release Management | • Initial release |

## Disclaimer

## 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 marketplace 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

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.

## 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.

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.

| Term | Description |
|---|---|
| Executable | Part of an Application. It consists of executable code (with exactly one entry point) created at integration time that can be deployed and installed on a Machine. An Application may consist of one or more Executables, each of which can be deployed to different Machines. |
| process | A process refers to the OS concept of a running process. **Attention:** process is **not equal** to Modelled Process (see below). Hence each Modelled Process has at some time a related (OS) process but a process may not always have a related Modelled Process. |
| Modelled Process | A Modelled Process is an instance of an Executable to be executed on a Machine and has a 1:1 association with the ARXML/Meta-Model element Process. This document also uses the term process (without the “modelled” prefix) to refer to the OS concept of a running process. |
| Reporting Process | A type of Modelled Process with an associated Executable where reportingBehavior is omitted ([TPS_MANI_01279]) or set to reportsExecutionState. A Reporting Process is expected to report its Execution State to Execution Management. |
| Non-reporting Process | A type of Modelled Process with an associated Executable where reportingBehavior set to doesNotReportExecutionState. A Non-reporting Process is **not** expected to report its Execution State to Execution Management. |
| Companion Process | A type of Reporting Process that is associated with Non-reporting Process and used to determine when functionality expected from Non-reporting Process is available. Whenever functional dependency exist on Non-reporting Process, you can configure proxy Execution Dependencies on Companion Process and make its own kRunning reporting conditional on monitored Non-reporting Process. |
| Self-terminating Process | A type of Modelled Process that has terminationBehavior configured to processIsSelfTerminating. This type of Modelled Process is allowed to self initiate termination procedure (i.e. just terminate with exit status EXIT_SUCCESS), or wait for Execution Management to initiate termination procedure via SIGTERM. |
| Unexpected Self-termination | The event consumed by Execution Management when a Modelled Process terminates without prior request via SIGTERM and has terminationBehavior configured to processIsNotSelfTerminating. Please note that every Unexpected Self-termination is also an Unexpected Termination, so requirements for the later apply here as well. |
| Unexpected Termination | The event consumed by Execution Management when a Modelled Process terminates with exit status other than 0 (EXIT_SUCCESS). Any kind of unhandled signal will result in an Unexpected Termination and thus a non 0 exit status. |

| | |
|---|---|
| Execution Dependency | Dependencies between Modelled Process instances can be configured to define a sequence for starting and terminating them. |
| Execution Management | The element of the AUTOSAR Adaptive Platform responsible for the ordered startup and shutdown of the AUTOSAR Adaptive Platform and Adaptive Applications. |
| State Management | The element defining modes of operation for AUTOSAR Adaptive Platform. 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 Modelled Processes which need to be controlled consistently. Depending on the state of the Function Group, processes (related to the Modelled Processes) are started or terminated.<br>Modelled Processes can belong to more than one Function Group State (but at exactly one Function Group).<br>"MachineFG" 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 element of State Management that characterizes the current status of a set of (functionally coherent) user-level Applications.<br>The set of Function Groups and their Function Group States is machine specific and are configured in Machine Manifest. |
| Undefined Function Group State | Any state of a Function Group, which is not modelled. A Function Group is in an Undefined Function Group State during state transition, if a state transition failed or if an Unexpected Termination or Unexpected Self-termination happened. |
| Machine State | A state of Function Group "MachineFG" with some predefined states (Startup/Shutdown/Restart). This can term can refer to the current state ("The Machine State is ..."), to a specific state ("In Machine State Startup ..."), or to a set of states ("In Machine States Startup or Shutdown ..."). |
| 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. |
| 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 Modelled 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). |
| DeterministicSyncMaster | A synchronization control point that receives the synchronization requests through a dedicated communication channel, for example ara::com, and sends the calculated cycle information for the next execution cycle to the connected DeterministicClients in the same domain. |

**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] Specification of the Adaptive Core
AUTOSAR_SWS_AdaptiveCore

[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 core specification [5] which is also applicable for `Execution Management`. The chapter “General requirements for all FunctionalClusters” of this specification shall be considered as an additional and required specification for implementation of `Execution Management`.

# 4 Constraints and assumptions

## 4.1 Known Limitations

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

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

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

Section 7.6.4 describes synchronization requirements for redundant deterministic execution that were required but not elaborated in 7.6.2. The interface of using a communication API other than `ara::com` is not in the scope of the specification. We focus on the single domain synchronization for the current release, i.e. the redundant deterministic execution is in the same OS or ECU. The models and configuration for deterministic synchronization and the details of interaction with Software Lockstep will be specified in later 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 7.7.3.2).

There are no requirements within this document that mandate a specific dependency the Persistency [8] functional cluster however an implementation of Execution Management may require the storage of persistent data. One possible use case might be Resource Group limitation of the data storage for processes.

Execution Management might provide supporting process information to Platform Health Management [9]. The exact form of the information is platform dependent and therefore not standardized by AUTOSAR. However it is expected to include information about process startup/termination and/or Function Group State change.

## 5.1 Protocol layer dependencies

None.

# 6 Requirements Tracing

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_00111]** | The AUTOSAR Adaptive Platform shall support source code portability for AUTOSAR Adaptive applications. | [SWS_EM_NA] |
| **[RS_AP_00114]** | C++ interface shall be compatible with C++14. | [SWS_EM_NA] |
| **[RS_AP_00115]** | Namespaces. | [SWS_EM_NA] |
| **[RS_AP_00116]** | Header file name. | [SWS_EM_02544] [SWS_EM_NA] |
| **[RS_AP_00119]** | Return values / application errors. | [SWS_EM_NA] |
| **[RS_AP_00120]** | Method and Function names. | [SWS_EM_02276] [SWS_EM_02283] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02290] [SWS_EM_02291] [SWS_EM_02542] |
| **[RS_AP_00121]** | Parameter names. | [SWS_EM_02276] [SWS_EM_02283] [SWS_EM_02288] [SWS_EM_02289] [SWS_EM_02291] [SWS_EM_02542] |
| **[RS_AP_00122]** | Type names. | [SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284] [SWS_EM_02541] [SWS_EM_02544] |
| **[RS_AP_00124]** | Variable names. | [SWS_EM_02544] [SWS_EM_02545] [SWS_EM_02546] [SWS_EM_NA] |
| **[RS_AP_00125]** | Enumerator and constant names. | [SWS_EM_NA] |
| **[RS_AP_00127]** | Usage of ara::core types. | [SWS_EM_02281] [SWS_EM_02282] [SWS_EM_02284] |
| **[RS_AP_00128]** | Error reporting. | [SWS_EM_02292] [SWS_EM_02542] |
| **[RS_AP_00129]** | Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation. | [SWS_EM_NA] |
| **[RS_AP_00130]** | AUTOSAR Adaptive Platform shall represent a rich and modern programming environment. | [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249] [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_02276] [SWS_EM_02283] [SWS_EM_02286] [SWS_EM_02287] [SWS_EM_02288] [SWS_EM_02290] [SWS_EM_02291] [SWS_EM_02542] [SWS_EM_NA] |
| **[RS_AP_00133]** | noexcept behavior of move and swap operations | [SWS_EM_NA] |

| Requirement | Description | Satisfied by |
|---|---|---|
| **[RS_AP_00134]** | noexcept behavior of class destructors | [SWS_EM_NA] |
| **[RS_AP_00135]** | Avoidance of shared ownership. | [SWS_EM_NA] |
| **[RS_AP_00136]** | Usage of string types. | [SWS_EM_NA] |
| **[RS_AP_00137]** | Connecting run-time interface with model. | [SWS_EM_NA] |
| **[RS_AP_00138]** | Return type of asynchronous function calls. | [SWS_EM_NA] |
| **[RS_AP_00139]** | Return type of synchronous function calls. | [SWS_EM_NA] |
| **[RS_AP_00140]** | Usage of "final specifier" in ara types. | [SWS_EM_02544] [SWS_EM_NA] |
| **[RS_AP_00141]** | Usage of out parameters. | [SWS_EM_NA] |
| **[RS_AP_00142]** | Handling of unsuccessful operations. | [SWS_EM_NA] |
| **[RS_AP_00143]** | Use 32-bit integral types by default. | [SWS_EM_NA] |
| **[RS_EM_00002]** | Execution Management shall set-up one process for the execution of each Modelled Process. | [SWS_EM_01014] [SWS_EM_01015] [SWS_EM_01041] [SWS_EM_01042] [SWS_EM_01043] |
| **[RS_EM_00005]** | Execution Management shall support the configuration of OS resource budgets for process and groups of processes. | [SWS_EM_02102] [SWS_EM_02103] [SWS_EM_02106] [SWS_EM_02107] [SWS_EM_02108] [SWS_EM_02109] |
| **[RS_EM_00008]** | Execution Management shall support the binding of all threads of a given process to a specified set of processor cores. | [SWS_EM_02104] |
| **[RS_EM_00009]** | Execution Management shall ensure it is the sole entity starting processes. | [SWS_EM_01030] [SWS_EM_01033] |
| **[RS_EM_00010]** | Execution Management shall support multiple instances of Executables. | [SWS_EM_01012] [SWS_EM_01072] [SWS_EM_01073] [SWS_EM_01074] [SWS_EM_01075] [SWS_EM_01076] [SWS_EM_01077] [SWS_EM_01078] [SWS_EM_02246] [SWS_EM_02247] [SWS_EM_02248] [SWS_EM_02249] |
| **[RS_EM_00011]** | Execution Management shall support self-initiated graceful shutdown of processes. | [SWS_EM_01404] [SWS_EM_01405] |
| **[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] |

| Requirement | Description | Satisfied by |
|---|---|---|
| **[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_02203] [SWS_EM_02210] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02216] [SWS_EM_02217] [SWS_EM_02510] [SWS_EM_02520] [SWS_EM_02530] [SWS_EM_02531] [SWS_EM_02532] [SWS_EM_02540] |
| **[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_01307] [SWS_EM_01308] [SWS_EM_01310] [SWS_EM_01311] [SWS_EM_01312] [SWS_EM_01313] [SWS_EM_01320] [SWS_EM_01321] [SWS_EM_01322] [SWS_EM_01323] [SWS_EM_01324] [SWS_EM_01325] [SWS_EM_01326] [SWS_EM_01327] [SWS_EM_02202] [SWS_EM_02203] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02220] [SWS_EM_02221] [SWS_EM_02225] [SWS_EM_02226] [SWS_EM_02230] [SWS_EM_02231] [SWS_EM_02235] [SWS_EM_02236] |
| **[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_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_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] [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_02280] [SWS_EM_02297] [SWS_EM_02298] [SWS_EM_02310] [SWS_EM_02311] [SWS_EM_02312] [SWS_EM_02313] [SWS_EM_02314] [SWS_EM_02541] [SWS_EM_02542] [SWS_EM_02543] [SWS_EM_02544] [SWS_EM_02545] [SWS_EM_02546] |

| Requirement | Description | Satisfied by |
|---|---|---|
| **[RS_EM_00103]** | Execution Management shall support process lifecycle management. | [SWS_EM_01002] [SWS_EM_01003] [SWS_EM_01004] [SWS_EM_01006] [SWS_EM_01055] [SWS_EM_01309] [SWS_EM_01314] [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] |
| **[RS_EM_00111]** | Execution Management shall assist identification of processes during Machine runtime. | [SWS_EM_02400] |
| **[RS_EM_00113]** | Execution Management shall support time-triggered execution. | [SWS_EM_01301] [SWS_EM_01310] [SWS_EM_01312] [SWS_EM_01313] |
| **[RS_EM_00150]** | Error Handling. | [SWS_EM_02032] [SWS_EM_02033] [SWS_EM_02034] |
| **[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_00111, RS_AP_00114, RS_AP_00115, RS_AP_00116, RS_AP_00119, RS_AP_00124, RS_AP_00125, RS_AP_00129, RS_AP_00132, RS_AP_00133, RS_AP_00134, RS_AP_00135, RS_AP_00136, RS_AP_00137, RS_AP_00138, RS_AP_00139, RS_AP_00140, RS_AP_00141, RS_AP_00142, RS_AP_00143, 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 Modelled Process. With the latter, we refer to an instance of the meta-model describing a process, it will eventually be realized by an operating system process.
- Section 7.3 covers the core Execution Management run-time responsibilities including the start of Applications.
- Section 7.4 describes the lifecycle of Applications including Modelled Process state transitions and startup / shutdown sequences.
- Section 7.5 covers several topics related to State Management within Execution Management including execution, Machine and Function Group state management.
- Section 7.6 documents support provided by Execution Management *Deterministic* execution such that given the same input and internal state, a calculation will always produce the same output.
- Section 7.7 describes how Execution Management supports resource management including the limitation of usage of CPU and memory by an Application.
- Section 7.8 provides an introduction to Fault Tolerance strategies in general. This section will be expanded in a future release to describe how such strategies are realized within Execution Management.
- Section 7.9 covers the topic of Trusted Platform, i.e. ensuring the integrity and authenticity of Applications.

## 7.1 Functional Cluster Lifecyle

### 7.1.1 Startup

See Section 7.5.2.1.

### 7.1.2 Shutdown

See Section 7.5.2.2.

### 7.1.3 Restart

See Section 7.5.2.2.

## 7.2 Technical Overview

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

### 7.2.1 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.

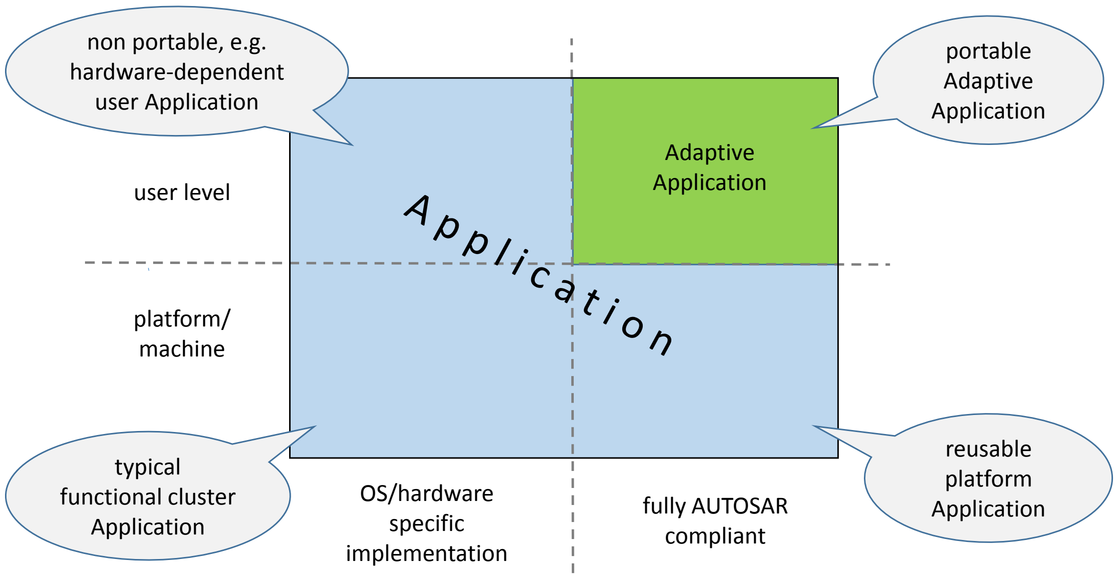
non portable, e.g. hardware-dependent user Application
portable Adaptive Application
user level
Adaptive Application
Application
platform/ machine
typical functional cluster Application
OS/hardware specific implementation
fully AUTOSAR compliant
reusable platform Application

**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 Machine. Previous version (if any) removed. | processed Manifests, stored in a platform-specific format which is efficiently readable at Machine startup. |
| Execution | process started as instance of the binary. | The Execution Management uses contents of the processed Manifests to start up and configure each process individually. |

**Table 7.1: Executable Lifecycle**

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

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

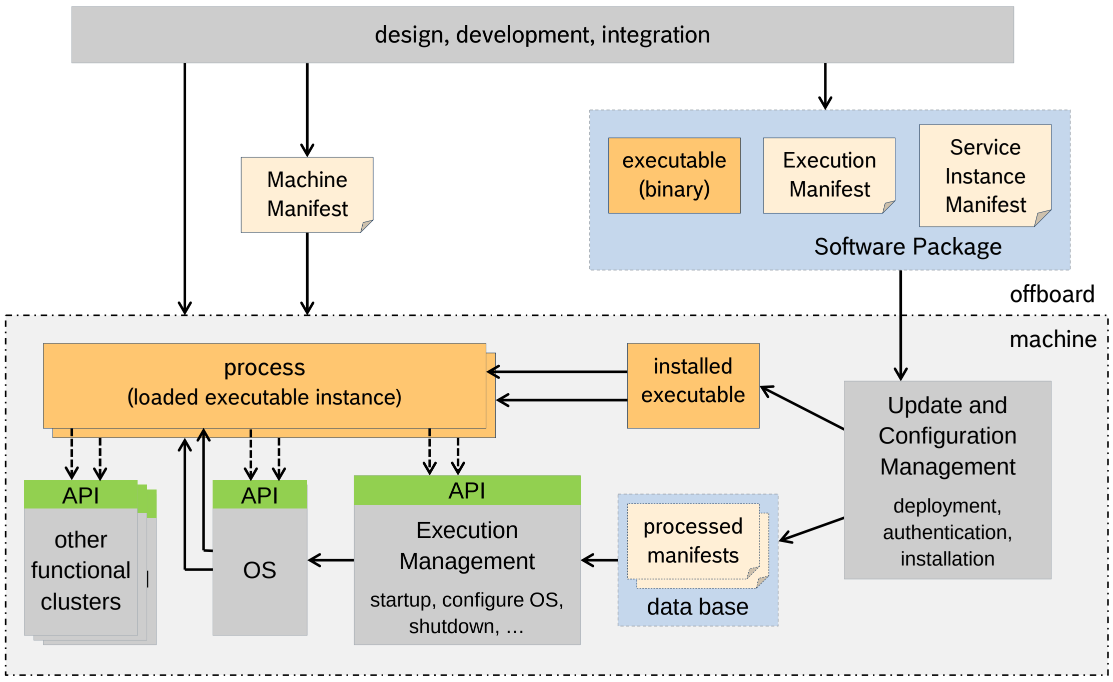
design, development, integration
Machine Manifest
executable (binary)
Execution Manifest
Service Instance Manifest
Software Package
offboard
machine
process (loaded executable instance)
installed executable
Update and Configuration Management
deployment, authentication, installation
API
other functional clusters
API
OS
API
Execution Management
startup, configure OS, shutdown, ...
processed manifests
data base

**Figure 7.2: Executable Lifecycle from deployment to execution**

### 7.2.4 Modelled Process

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

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

**Remark:** In this release of this document it is mostly assumed that processes are self-contained, i.e. that they take care of controlling thread creation and scheduling by calling APIs of the Operating System Interface from within the code. Execution Management only starts and terminates the processes and while the processes are running, Execution Management only interacts with the processes by providing State Management mechanisms (see 7.5) or APIs to support Deterministic Execution (see 7.6.3).

### 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 Modelled Process properties (startup parameters, resource group assignment, scheduling priorities etc.).

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

Each instance of an Executable binary, i.e. each started process, is individually configurable, with the option to use a different configuration set per Machine State or per Function Group State (see Section 7.5 and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_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 Modelled Processes), i.e. which is not already covered by an Execution Manifest or a Service Instance Manifest.

The contents of a Machine Manifest includes the configuration of Machine properties and features (resources, safety, security, etc.), 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 7.9.

**[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.

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

## 7.4 Process Lifecycle Management

### 7.4.1 Execution State

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

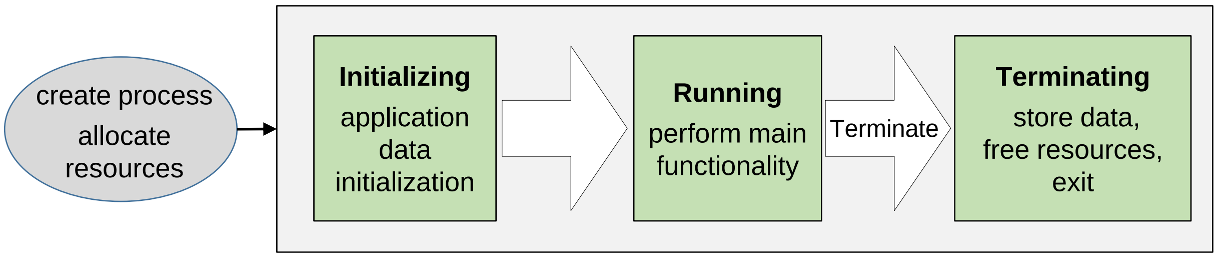
create process
allocate resources
Initializing
application data initialization
Running
perform main functionality
Terminate
Terminating
store data, free resources, exit

**Figure 7.3: Execution States**

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

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.

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

**[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.

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

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

On receipt of SIGTERM, a process simply commences the actual termination.

During the Terminating state, the process is expected to save persistent data and free all internally used resources. The process indicates completion of the Terminating state by termination with exit status 0 (EXIT_SUCCESS).

Execution Management 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.

**[SWS_EM_01314]**{DRAFT} **Default value for terminationBehavior** ⌈Execution Management shall treat Modelled Processes without terminationBehavior configuration not as Self-terminating Processes. Such Modelled Processes are expected to terminate only on Execution Management request.⌋ *(RS_EM_00103)*

**[SWS_EM_01309]**{DRAFT} **Unexpected Termination of a process** ⌈In case of Unexpected Termination, Execution Management shall perform the following actions:

1. Log event if needed
2. Set the current Function Group State to Undefined Function Group State.
3. Report the configured executionError via the ara::exec::StateClient::GetExecutionError interface.

⌋ *(RS_EM_00103)*

Please note that [SWS_EM_01309] also applies for Unexpected Self-termination.

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

**[SWS_EM_02243]**{DRAFT} **Handling Execution State Running** ⌈Execution Management shall return kInvalidTransition when a process reports Execution State kRunning and the process is not in Process State Initializing.⌋ *(RS_EM_00103)*

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

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

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

**[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 ResourceGroup can be used (see Section 7.7).

An attempt to report *Execution State* by a Non-reporting Process is considered an error by Execution Management, see [SWS_EM_01403].

### 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. Process State is used by Execution Management to resolve Execution Dependencies, manage timeouts, etc.

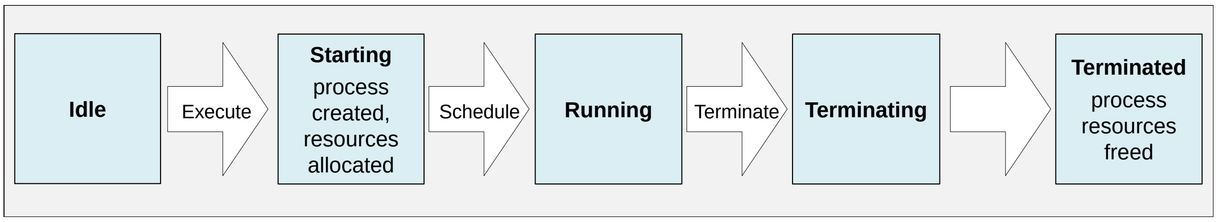

**Figure 7.4: process Lifecycle**

**[SWS_EM_01401] process Self Reporting** ⌈The AUTOSAR Adaptive Platform 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]**{OBSOLETE} **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)*

**[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]. The distinction between Process State Idle and Terminated is relevant for resolving Execution Dependencies to Self-terminating Processes (see Section 7.4.3.1).

### 7.4.3 Startup and Termination

#### 7.4.3.1 Execution Dependency

Execution Management can derive an ordering for the startup and termination of processes within State Management framework based on the declared Execution Dependencies. This ensures that Applications are started before dependent Applications use the services that they provide and, likewise, that Applications are shutdown only when their provided services are no longer required.

The Execution Dependencies, see [TPS_MANI_01041] and [constr_1606], are configured in the Execution Manifests, which is created at integration time based on information provided by the Application developer. An Execution Dependency defines the provider of functionality required by a process necessary for that process to provide its own functionality. Execution Management ensures the dependent processes are in the state defined by the Execution Dependency before the process defining the dependency is started.

User-level Applications are expected to use the service discovery mechanisms of Communication Management as the primary mechanism for execution sequencing as this is supported both within a Machine and across Machine boundaries. Thus user-level applications should not rely on Execution Dependencies unless strictly necessary. Which processes are running depends on the current Function Group States, including the Machine State, see Section 7.5. The integrator should ensure that all service dependencies are mapped to 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. See also [constr_1744] in [4].

**Example 7.1**

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

processes are only started by Execution Management if they reference a requested Machine State or Function Group State, but not because of configured Execution Dependencies. Execution Dependencies are only used to control a startup or terminate sequence at state transitions. Note that the scope of Execution Dependency resolution is limited to one Function Group State only (see [constr_1689] and [SWS_EM_02245]).

**[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 termination $Storage$ after $DataLogger$ so the latter can flush its data during termination.

Note that [SWS_EM_01051] merely requires Execution Management to not terminate the dependent processes before terminating a process. It is not an error if the process has self-terminated so is not available to be terminated.

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

**Example 7.3**

Consider three processes:

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

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

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

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

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

**Example 7.4**

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

Please note that requirement [SWS_EM_01001] effectively forbids any Execution Dependencies that spans outside of a single Function Group State (or a Machine State) definition, see also [constr_1689]. 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 Process State Running directly after start. Regardless of whether the process has completed its initialization phase and is ready to offer its services or not. This means that Running Execution Dependencies are immediately satisfied and this do not achieve the original semantics when specified for a Non-reporting Processes without further action.

This limitation can be overcome by introducing a Companion Process, which acts as a representative of the Non-reporting Process. The Companion Process waits for availability of the service provided by the Non-reporting Process and reports kRunning to Execution Management. The processes which in fact need the services of the Non-reporting Process can be configured to be dependent on the Companion Process. Please note that the Terminated Execution Dependency is not affected as Execution Management is informed by the Operating System when Non-reporting Processes are Terminated. Please see Figure 7.5 for more details.

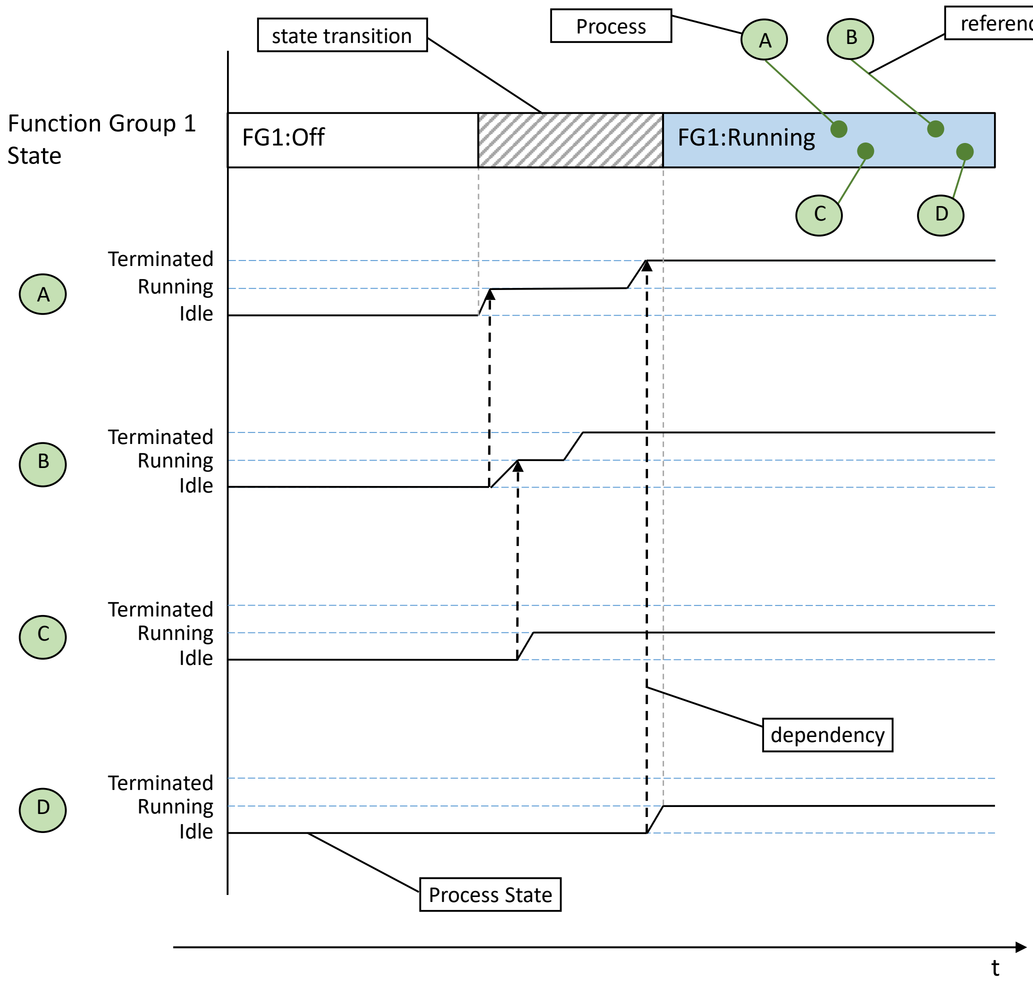

**Figure 7.5: Execution dependencies on Non-reporting Process**

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

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

Please note this approach can also be used to communicate a Health State to Platform Health Management.

#### 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 ProcessArgument of the StartupConfig referenced by the StateDependentStartupConfig shall be passed to the process by Execution Management based on [SWS_EM_01072] and [SWS_EM_01078].⌋*(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 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 passing arguments to a process in the same way that a shell passes command line arguments to a POSIX process. Execution Management assigns each ProcessArgument.argument to an element in the argv[] array, starting at element index 1, and passes this to the process main() function. ProcessArgument ordering is used to preserve the semantics of an (option, argument) pair such as "-b value", where the "-b" argument must precede the "value" argument. This method supports the short form and long form argument passing conventions typically used in POSIX environments.

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

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

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

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

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

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

**[SWS_EM_01078]**{DRAFT} **Process Argument strings** ⌈ProcessArgument.argument shall be passed to the process in order with the first ProcessArgument.argument starting at Process Argument 1.⌋*(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 has 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 Machine Startup Sequence

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

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

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

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

**[SWS_EM_01000] Startup order** ⌈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 7.6 shows the overall startup sequence.

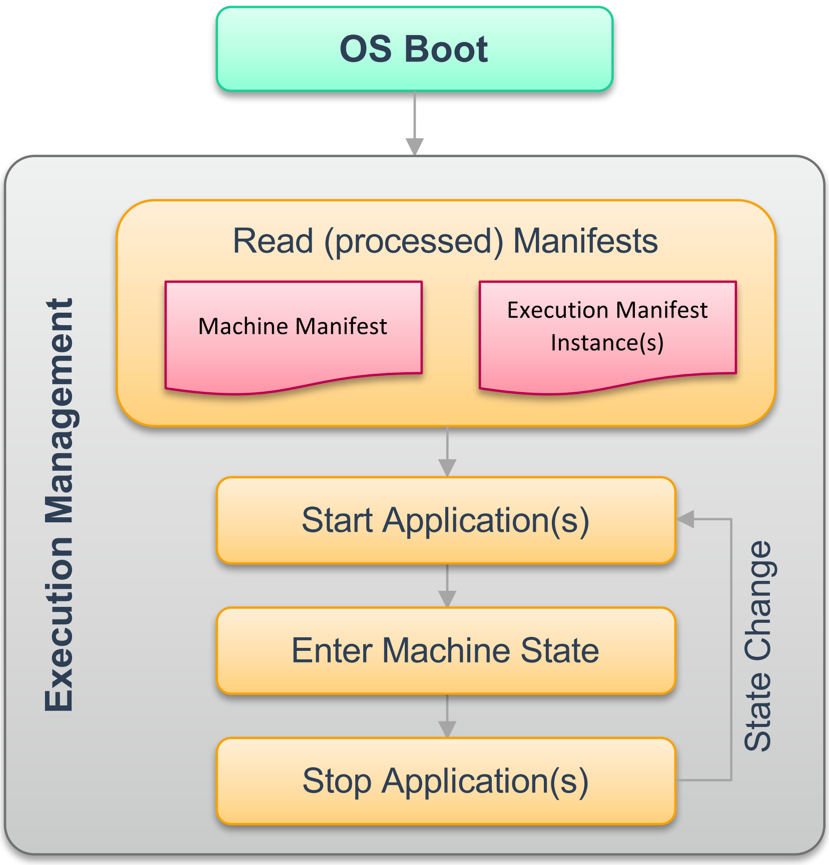

**Figure 7.6: Startup sequence**

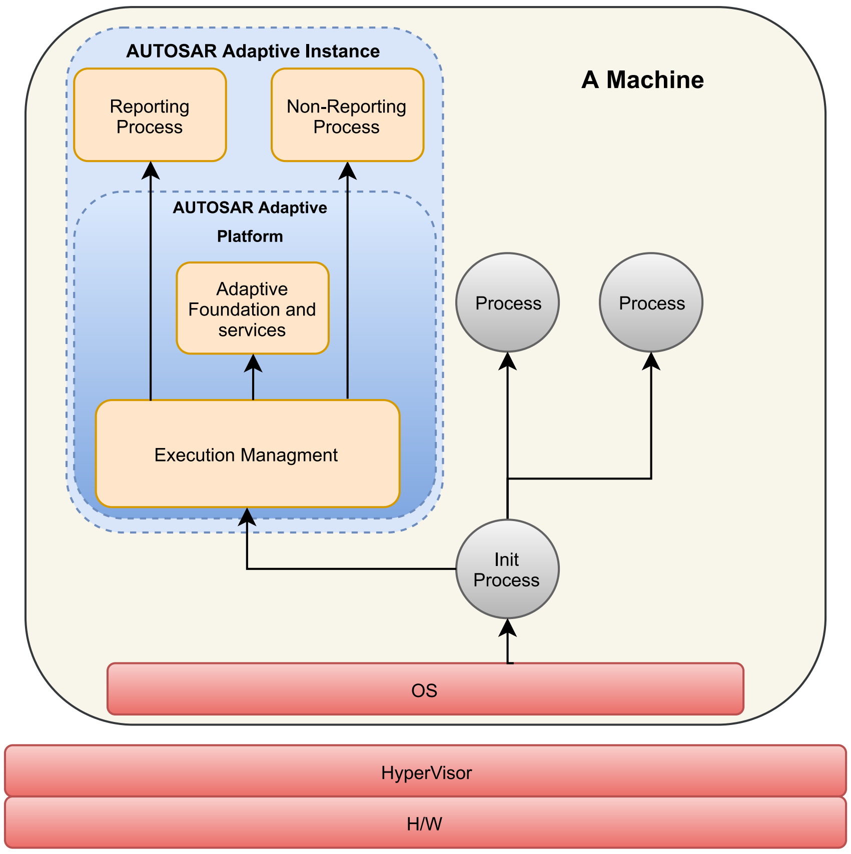

**Figure 7.7: AUTOSAR Adaptive Platform Boundary**

## 7.5 State Management

### 7.5.1 Overview

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

The Execution Manifest allows to define in which states the Modelled Processes have to run (see [4]). As mentioned before, a Modelled 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 "MachineFG".

The Function Group "MachineFG" has several mandatory states (see [SWS_EM_02250]) 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 of "MachineFG"), 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 "MachineFG" 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 7.8.

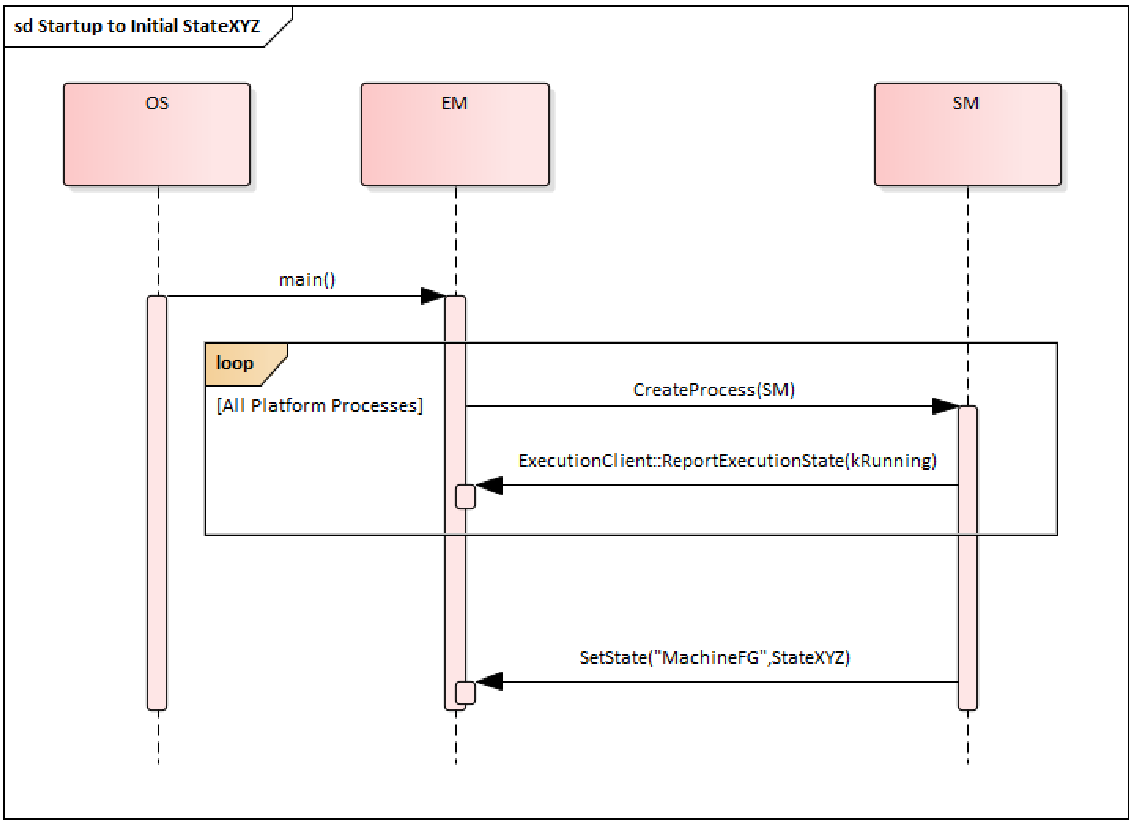
sd Startup to Initial StateXYZ
OS
EM
SM
main()
loop
[All Platform Processes]
CreateProcess(SM)
ExecutionClient::ReportExecutionState(kRunning)
SetState("MachineFG",StateXYZ)

**Figure 7.8: Start-up Sequence – from `Startup` to initial running state `StateXYZ`**

An arbitrary state change sequence to machine state StateXYZ is illustrated in Figure 7.9. 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.

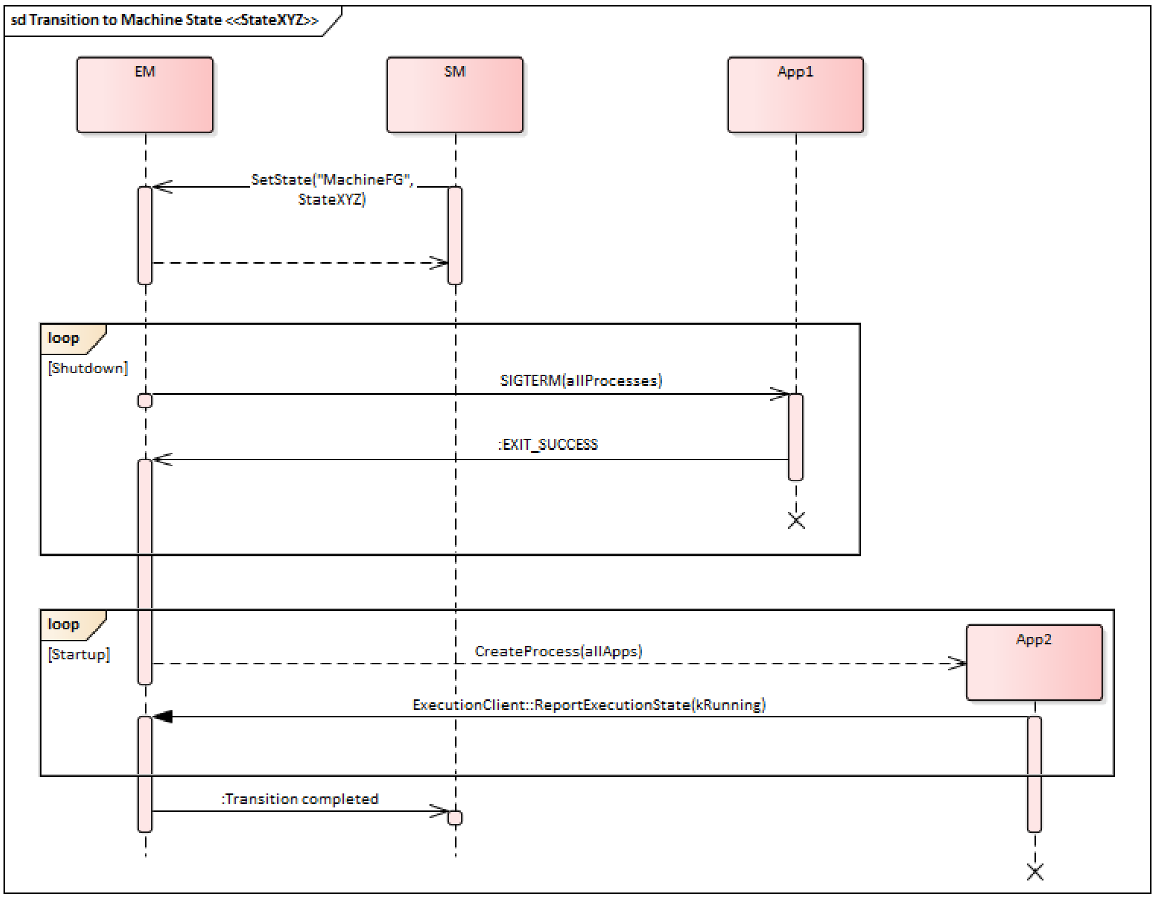
sd Transition to Machine State <<StateXYZ>>
EM
SM
App1
SetState("MachineFG", StateXYZ)
loop
[Shutdown]
SIGTERM(allProcesses)
:EXIT_SUCCESS
loop
[Startup]
App2
CreateProcess(allApps)
ExecutionClient::ReportExecutionState(kRunning)
:Transition completed

**Figure 7.9: State Change Sequence – Transition to machine state `StateXYZ`**

#### 7.5.2.1 Startup

**[SWS_EM_02250]**{DRAFT} **Machine State Startup** ⌈`Execution Management` shall cease `AUTOSAR Adaptive Platform` startup if the `Startup` state is not configured for `Function Group` `"MachineFG"`.⌋*(RS_EM_00101)*

There are multiple possible strategies after cessation; halting (e.g. in an endless loop), aborting (e.g. resetting ECU through watchdog), etc. The choice is implementation-specific.

**[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)*

After the Startup state is reached (as described by [SWS_EM_02241]) Execution Management does not initiate any further Function Group State changes (this includes Machine State). Instead such changes are requested by State Management and then performed by Execution Management.

Execution Management will be controlled by other software entities and should not execute any Function Group State changes on its own (with one exception: [SWS_EM_01023]). This creates some expectations towards system configuration. The specification expects that State Management will be configured to run in every Machine State (this includes Startup, Shutdown and Restart). 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/Restart

Execution Management does not perform shutdown/restart 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 the relevant Machine State. See [4] [constr_1618] and [constr_1619].

[constr_1687] mandates specification of both Shutdown / Restart Machine State. Execution Management can check that the state is configured for a Function Group "MachineFG" and take implementation-specific action, e.g. log, if desired however Platform startup can still proceed.

A request to Execution Management to change the current Machine State to either Shutdown or Restart is handled the same as any other Function Group state change request. From the point of view of Execution Management all Function Groups are independent and therefore changes to them, can be applied without any side effects.

However, from the point of view of State Management, where knowledge of the dependencies between different Function Groups exist this may not be true. AUTOSAR assumes that State Management will requests "MachineFG" Shutdown or Restart when it's valid to do so; see [11] for advice on how to orchestrate shutdown of the Machine.

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

As mentioned in Section 7.5.2.1, AUTOSAR assumes that State Management will be configured to run in Shutdown and Restart. State transition is not a trivial system change and it can fail for a number of reasons - in which case State Management should remain alive so you can report an error and wait for further instructions. Please note that very purpose of this state is to shutdown or restart the 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.

### 7.5.3 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 7.10 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.

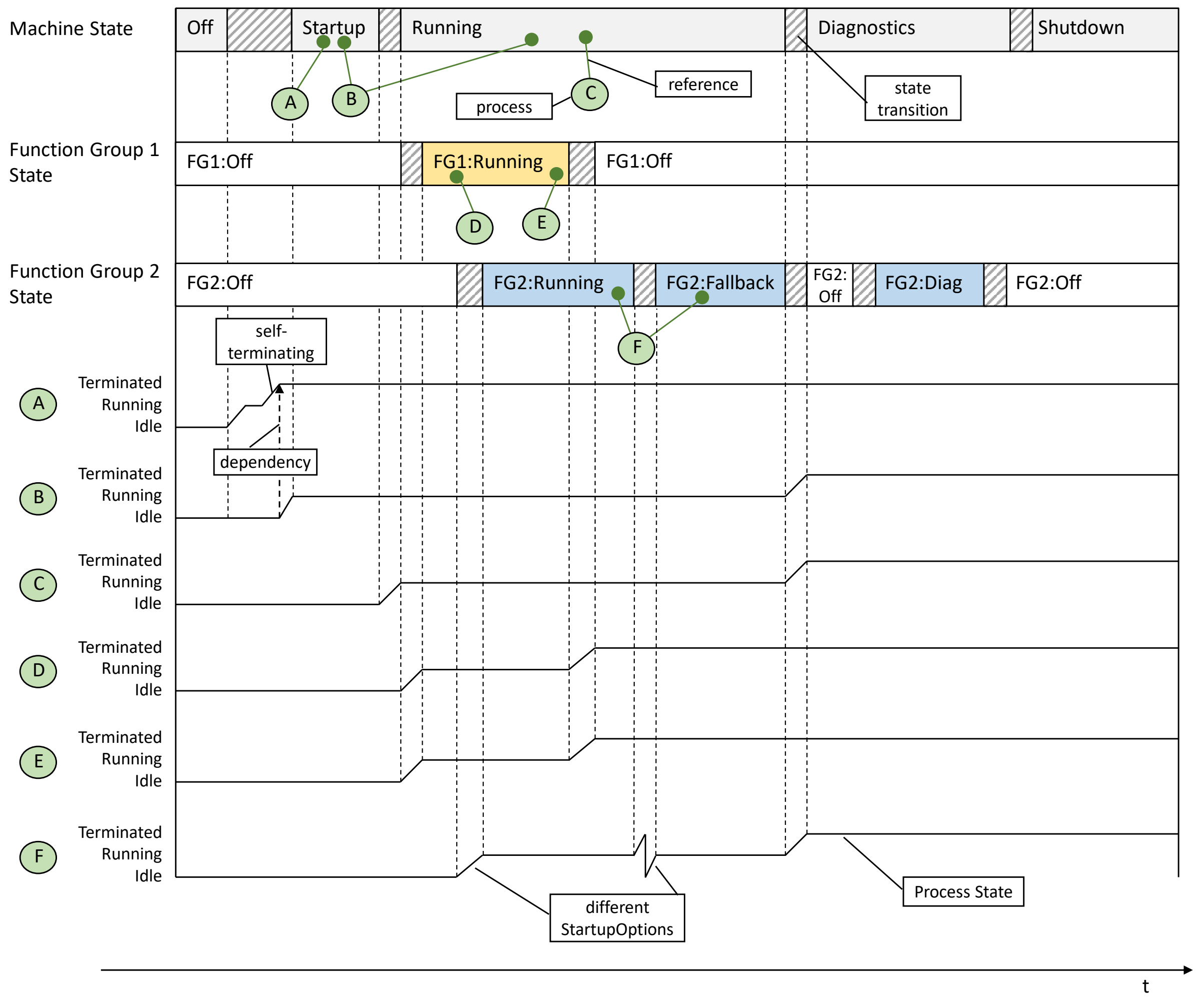

**Figure 7.10: State dependent process control**

- 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 7.4.3.1
- 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 Modelled Process, depending on the current Function Group State and on information provided in the Execution Manifests.⌋*(RS_EM_00101)*

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

**[SWS_EM_01033]**{DRAFT} **process start-up configuration** ⌈To enable a Modelled Process to be launched in multiple Function Group States, Execution Management shall be able to configure the Modelled 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 "MachineFG") 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.

The "Off" state is mandatory [TPS_MANI_03195] and must not have Modelled Processes mapped [constr_3424].

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 7.9 applies to state changes of any Function Group - just replace "MachineFG" by the name of the Function Group. On receipt of the state change request, Execution Management terminates no longer needed processes and then starts processes active in the new Function Group State before confirming the state change to State Management. For details see Section 7.5.5.

### 7.5.4 State Interaction

Figure 7.11 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.

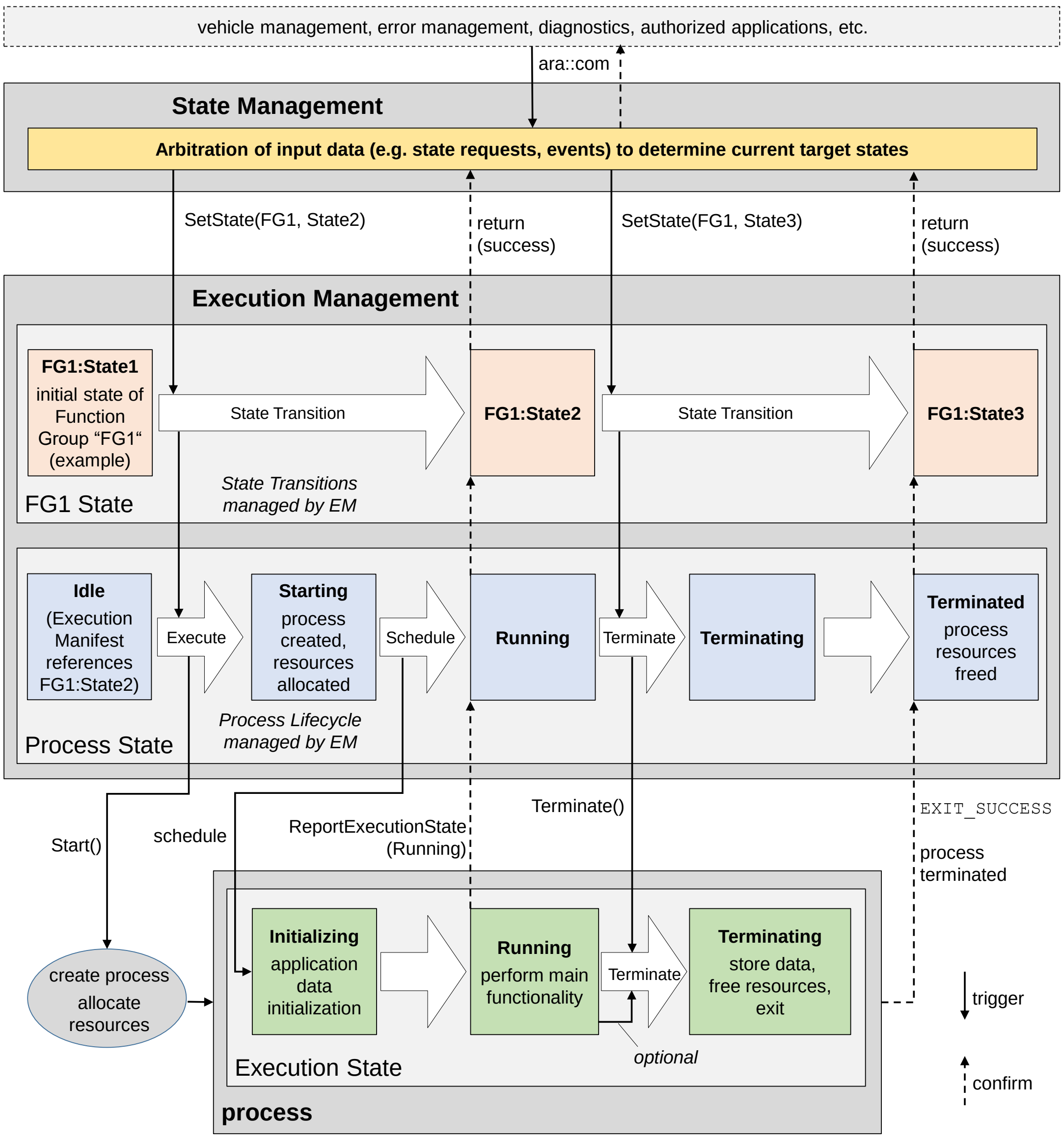
vehicle management, error management, diagnostics, authorized applications, etc.
ara::com
State Management
Arbitration of input data (e.g. state requests, events) to determine current target states
SetState(FG1, State2)
return (success)
SetState(FG1, State3)
return (success)
Execution Management
FG1:State1 initial state of Function Group "FG1" (example)
State Transition
FG1:State2
State Transition
FG1:State3
FG1 State
State Transitions managed by EM
Idle (Execution Manifest references FG1:State2)
Execute
Starting process created, resources allocated
Schedule
Running
Terminate
Terminating
Terminated process resources freed
Process State
Process Lifecycle managed by EM
Start()
schedule
ReportExecutionState (Running)
Terminate()
EXIT_SUCCESS
process terminated
create process allocate resources
Initializing application data initialization
Running perform main functionality
Terminate
Terminating store data, free resources, exit
optional
Execution State
process
trigger
confirm

**Figure 7.11: 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 8.2.7. 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: "MachineFG".

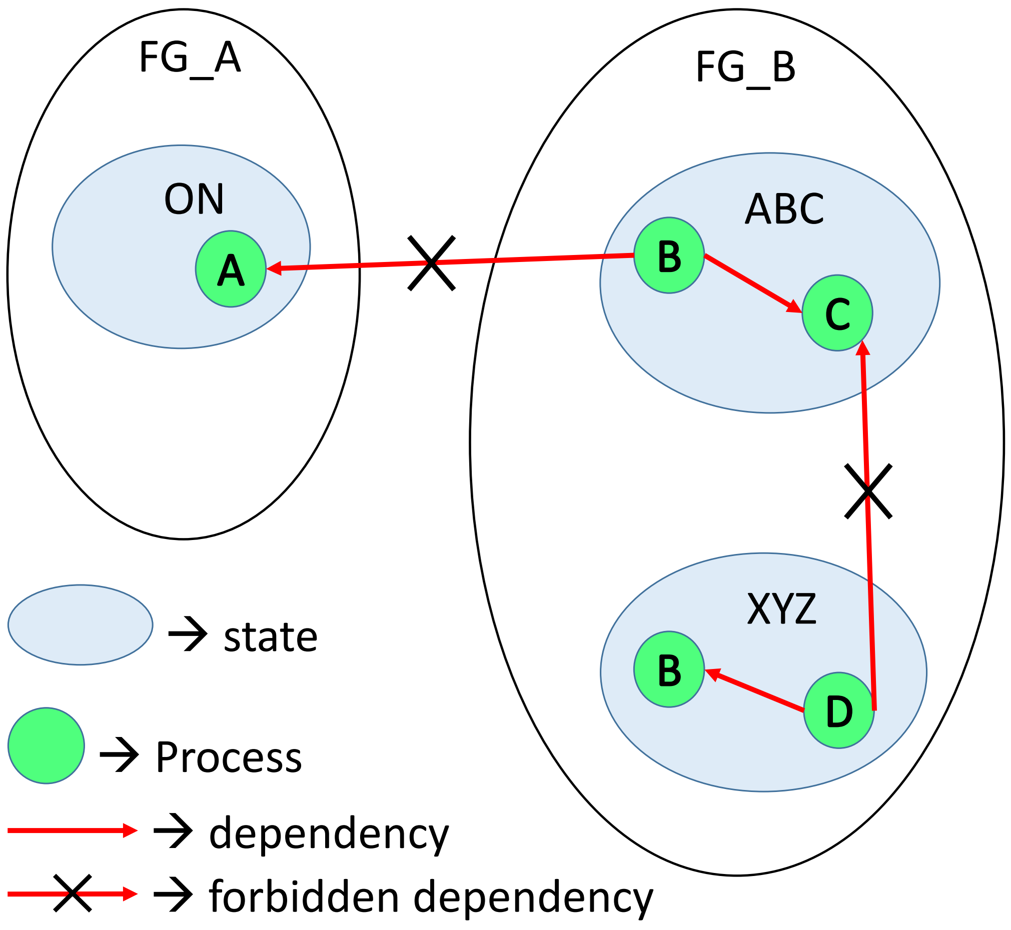

**Figure 7.12: Example configuration for state transition**

Before we specify how internals of a state transition works, let's consider an example configuration illustrated in figure Figure 7.12. 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 7.4.3.1 ([SWS_EM_01001] and [constr_1689]).

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

From the above we can conclude that each Function Group is a separate entity and state transition of one Function Group doesn't have side effects on another Function Group. Please note that this is true from the point of view of Execution Management and may differ from the point of view of State Management (see [11] if you need more information on this).

In the following requirements, the term "the process references a State" means that a Modelled 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 "MachineFg" name. In short this is a Function Group State or Machine State.

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

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

StartupConfig 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 reference the CurrentState in its Execution Manifest, but don't reference 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:

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

⌋*(RS_EM_00101)*

**[SWS_EM_02311]**{DRAFT} **Order of process termination timeout reaction** ⌈Execution Management shall perform the terminate reaction [SWS_EM_02255] before reporting to State Management [SWS_EM_02258].⌋*(RS_EM_00101)*

**[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 startup (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 attempt to restart the process up to numberOfRestartAttempts times.⌋*(RS_EM_00101)*

process start-up timeout is caused by a malfunction and therefore Execution Management requests termination of the process by the operating system (e.g. using SIGKILL) rather than requesting termination through SIGTERM as the process is assumed to be in an erroneous state.

**[SWS_EM_02280]**{DRAFT} **Effect on Execution Dependency** ⌈A restart attempt according to [SWS_EM_02260] shall not fulfill any terminated dependencies.⌋*(RS_EM_00101)*

**[SWS_EM_02310]**{DRAFT} **State transition - process termination after start-up timeout reaction** ⌈In case a process start-up timeout occurred after Execution Management attemted to restart the process numberOfRestartAttempts times, Execution Management shall request the Operating System to terminate the underlying process.⌋*(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:

1. Stop the Function Group State transition, so State Management can decide how to proceed.
2. Log event if needed

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

⌋*(RS_EM_00101)*

**[SWS_EM_02312]**{DRAFT} **Order of process start-up timeout reaction** ⌈Execution Management shall perform the terminate reaction [SWS_EM_02310] before reporting to State Management [SWS_EM_02259].⌋*(RS_EM_00101)*

When starting new processes, Execution Management is obligated to perform dependency resolution. When doing so it may came across a configuration where process B depends on process A, but process A needs to be restarted during state change. Another example is a configuration where process D depends on a Self-terminating Process C to be in Process State Terminated. process C has to be started and terminated in the requested Function Group State to fulfill D's Execution Dependency. Please see Figure 7.13 for more details.

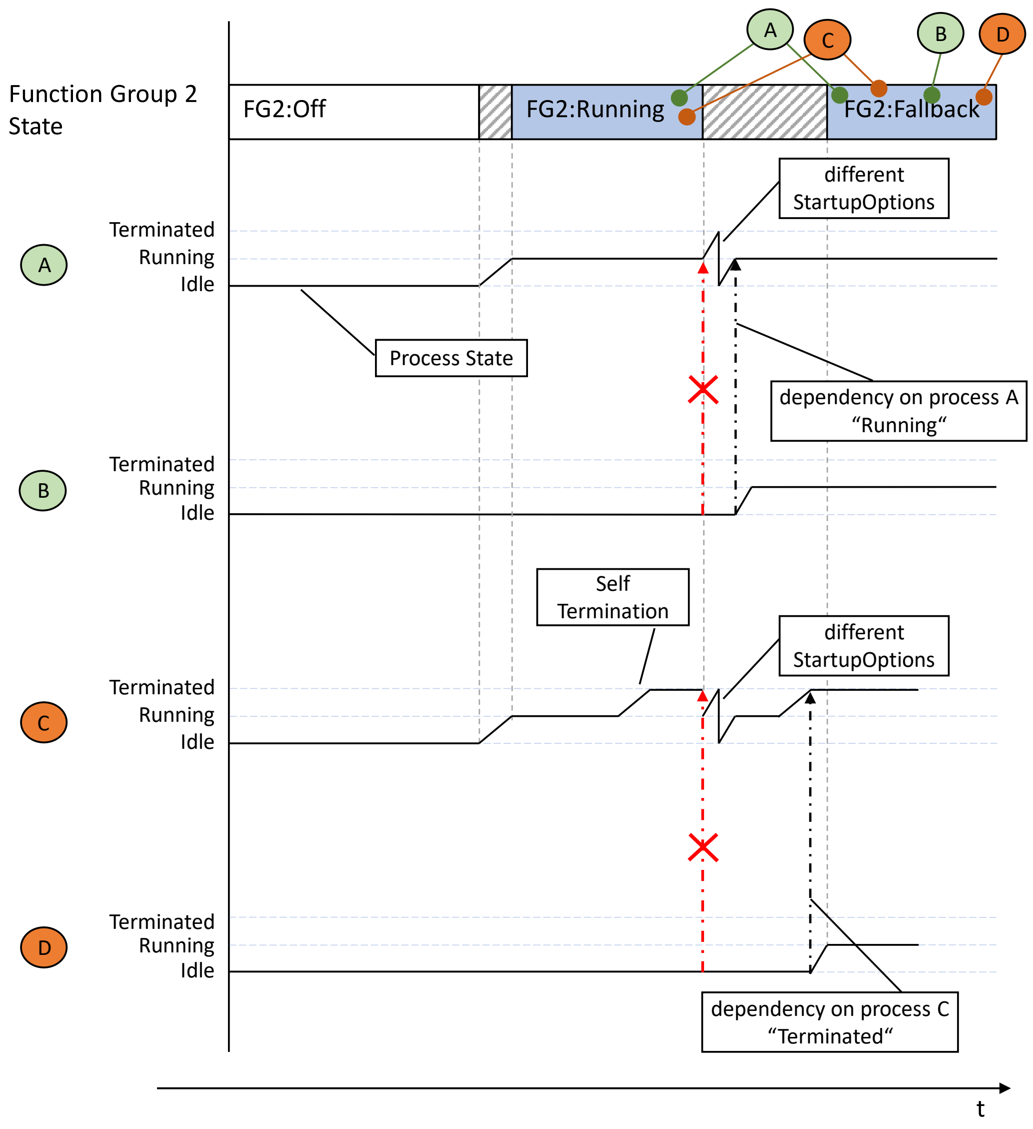

**Figure 7.13: Dependency resolution during state change**

**[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.

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

- Dependency resolution ([SWS_EM_02245]) has identified processes to start/stop
- All processes expected to terminate have terminated ([SWS_EM_01060])
- All started ([SWS_EM_01066]) or restarted [SWS_EM_02251]) Reporting Processes have reported kRunning.

**[SWS_EM_01067]**{DRAFT} **Actions on Completion State Transition** ⌈On successful completion of a state transition, Execution Management shall set the CurrentState to the RequestedState and report success back to State Management.⌋*(RS_EM_00101)*

**[SWS_EM_02313]**{DRAFT} **Unexpected Termination of starting processes during Function Group State transition** ⌈In case of Unexpected Termination during process startup ([SWS_EM_01066]), Execution Management shall perform the following actions:

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

⌋*(RS_EM_00101)*

**[SWS_EM_02314]**{DRAFT} **Unexpected Termination of terminating processes during Function Group State transition** ⌈In case of Unexpected Termination during process termination ([SWS_EM_01060],[SWS_EM_02251]), Execution Management shall perform the following actions:

1. Stop the Function Group State transition, so State Management can decide how to proceed.
2. Log event if needed

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

⌋*(RS_EM_00101)*

**[SWS_EM_02297]**{DRAFT} **StateClient usage restriction** ⌈`StateClient` API shall return `kCommunicationError` when invoked by a `process` with `Process.functionClusterAffiliation` configured to anything else than STATE_MANAGEMENT.⌋*(RS_EM_00101)*

If not protected `StateClient` can be used to destabilise `Machine`, see Section 8.2.7 for more details.

## 7.6 Deterministic Execution

### 7.6.1 Determinism

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

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

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

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

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

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

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

- Software Lockstep: To execute ASIL C/D applications with high computing performance demands, specific measures, such as software lockstep are required, due to high transient hardware error rates of high performance microprocessors. Software lockstep is a technique where the calculation is done redundantly through two different execution paths and the results are compared. To make the redundant calculations comparable, software lockstep requires a fully deterministic calculation. For details see 7.6.2.
- Reuse of verified software: The deterministic subsystem shows the same behavior on different platforms which satisfy the performance and resource needs of the subsystem, regardless of other differences in each environment, such as existence of unrelated applications. Examples include the different development and simulation platforms. Due to reproducible functional behavior, many results of testing, configuration and calibration of the subsystem are valid in each environment where the subsystem is deployed on and don’t need to be repeated.

#### 7.6.1.1 Time Determinism

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

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

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

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

#### 7.6.1.2 Data Determinism

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

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

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

#### 7.6.1.3 Full Determinism

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

Currently, Full Deterministic behavior of one `process` is supported. Determinism of a cluster of `Modelled Processes` on one machine is outlined in the document [12].

Determinism over several machines needs extensions of Communication Management which have not been specified yet.

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

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

### 7.6.2 Deterministic Client

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

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

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 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 considers how DeterministicClient can be used in one of the redundantly executed processes.

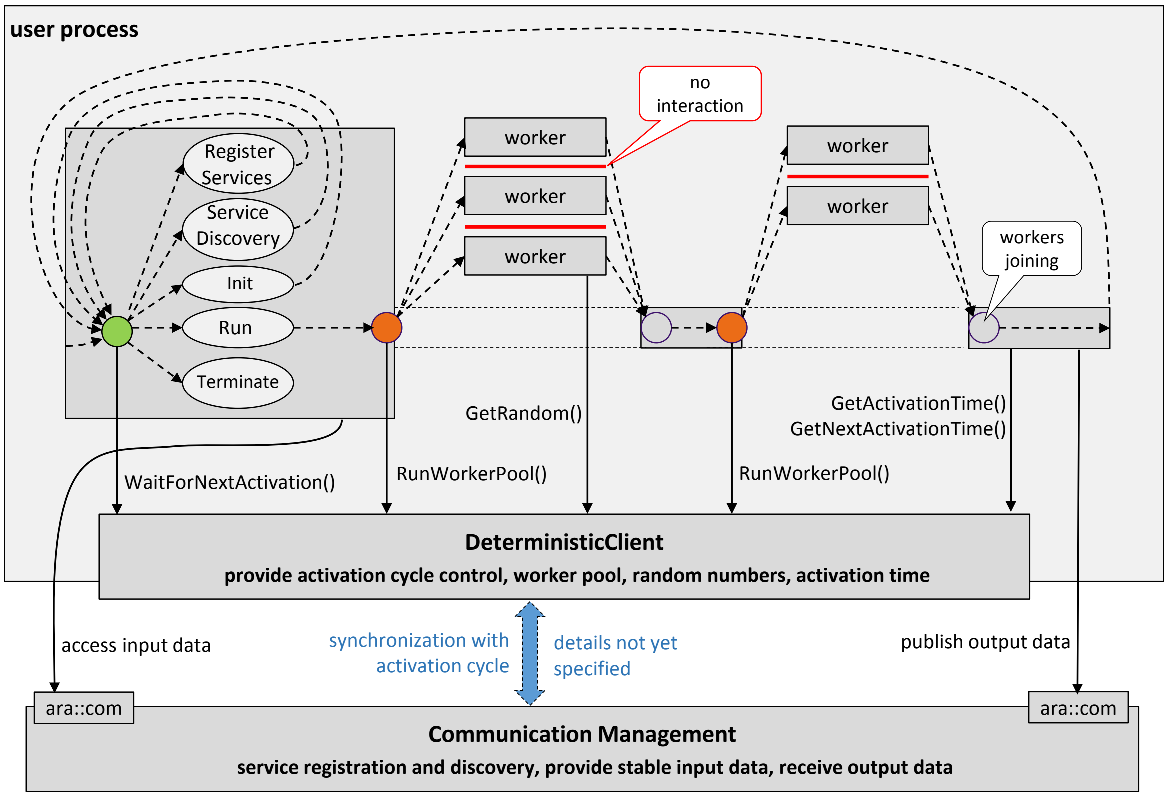

**Figure 7.14: Cyclic Deterministic Execution**

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

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

Additional `DeterministicClient` APIs provide random numbers and 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 7.6.3.3 and 7.6.3.4.

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

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

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.

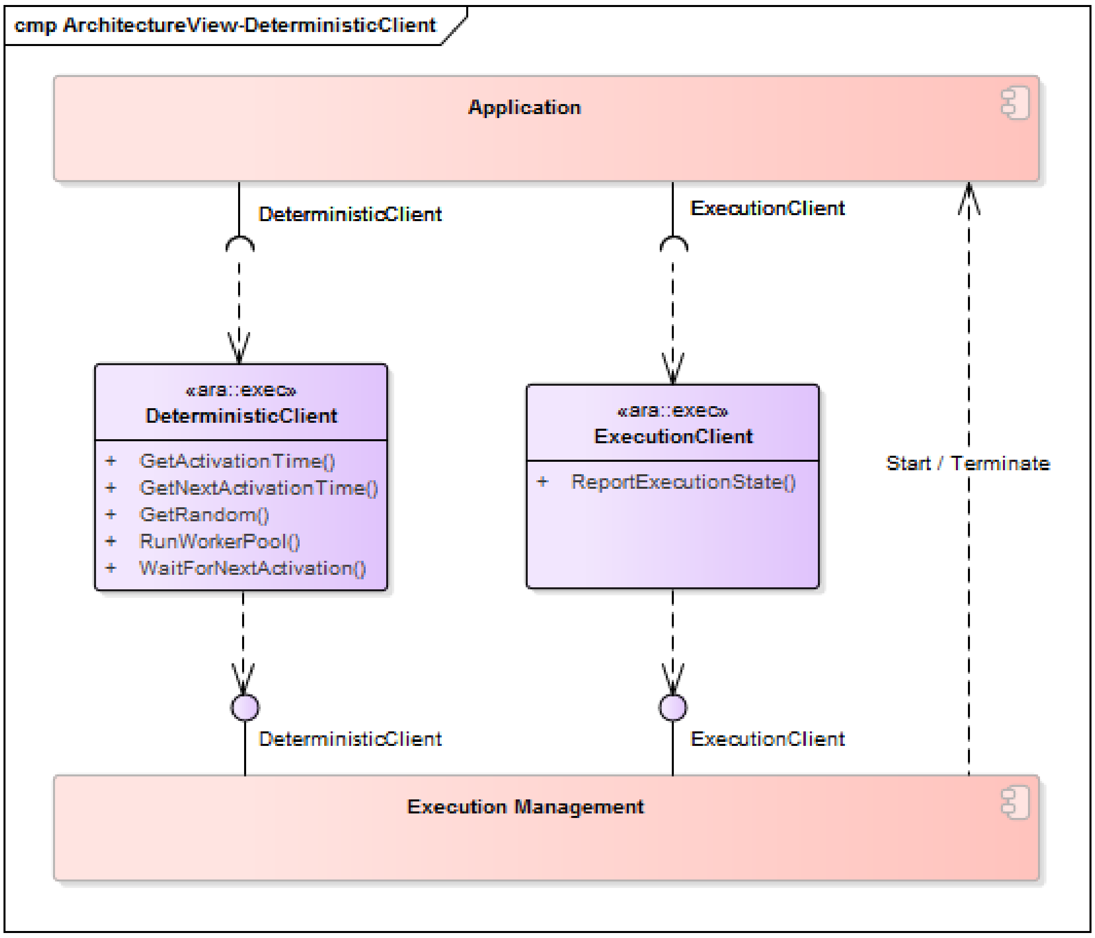

**Figure 7.15: Deterministic Execution Interface**

### 7.6.3 Cyclic Deterministic Execution

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

#### 7.6.3.1 Control of Cyclic Execution

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

**[SWS_EM_01301] Cyclic Execution** ⌈`Execution Management` shall provide a blocking wait point API `ara::exec::DeterministicClient::WaitForActivation`.⌋*(RS_EM_00052, RS_EM_00113)*

After the `process` has been started by `Execution Management`, it reports `ara::exec::ExecutionState kRunning` (see 7.4.1) and calls `ara::exec::DeterministicClient::WaitForActivation`.

The process executes one cycle when ara::exec::DeterministicClient::WaitForActivation 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** ⌈ara::exec::DeterministicClient::WaitForActivation 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 ara::exec::ActivationReturnType returned from ara::exec::DeterministicClient::WaitForActivation determines the actions taken at each cycle:

- kRegisterServices – 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.
- kServiceDiscovery – 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.
- kInit – The process initializes its internal data structures. The worker pool (see 7.6.3.2) can be accessed once or several time sequentially. A budget (see 7.6.3.5) needs to be assigned to the “Init” cycle.
- kRun – The process performs one cycle of its normal cyclic execution. This can be repeated indefinitely. The worker pool (see 7.6.3.2) can be accessed once or several times sequentially within a cycle. A budget (see 7.6.3.5) needs to be assigned.
- kTerminate – The process prepares to terminate. The actual termination is performed according to [SWS_EM_01404], see section 7.4.2.

**[SWS_EM_01303] Cyclic Execution Control Sequence** ⌈The return code of ara::exec::DeterministicClient::WaitForActivation shall follow this sequence: kRegisterServices, kServiceDiscovery, kInit, kRun, and kTerminate. Note that kRun is expected to be returned multiple times.⌋*(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, ara::exec::DeterministicClient::WaitForActivation shall return kServiceDiscovery once immediately after ara::exec::DeterministicClient::WaitForActivation is called, and then continue with the normal kRun cycle.⌋*(RS_EM_00052)*

The service discovery update needs to be triggered in an implementation specific way, e.g. through a callback triggered by StartFindService indicating service unavailability. 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 ara::exec::DeterministicClient::WaitForActivation 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 ara::exec::DeterministicClient::WaitForActivation returns periodically based on a defined period.
- Event-triggered activation means that ara::exec::DeterministicClient::WaitForActivation 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 of the synchronization for both periodic and event-triggered activation are discussed in section 7.6.4.

**[SWS_EM_01351] Execution Cycle Time** ⌈ara::exec::DeterministicClient::WaitForActivation shall return with kRun when a configurable cycleTimeValue 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 ara::exec::DeterministicClient::WaitForActivation within a kRun cycle after the configured 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 cycleTimeValue is zero, ara::exec::DeterministicClient::WaitForActivation 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.

#### 7.6.3.2 Worker Pool

**[SWS_EM_01305] Worker Pool** ⌈Execution Management shall provide a blocking API ara::exec::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. ara::exec::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.

ara::exec::DeterministicClient::RunWorkerPool registers a “worker” runnable object, along with its parameter object. The container 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** ⌈ara::exec::DeterministicClient::RunWorkerPool shall sequentially (using the iterator of input parameter container) call a method ara::exec::WorkerRunnable::Run (input parameter runnableObj) on every element of container, by using a worker pool of size numberOfWorkers.⌋*(RS_EM_00053)*

**[SWS_EM_01307]**{DRAFT} **Worker Object** ⌈The Worker object passed to ara::exec::DeterministicClient::RunWorkerPool shall be derived from ara::exec::WorkerRunnable using public inheritance.⌋*(RS_EM_00053)*

Within ara::exec::DeterministicClient::RunWorkerPool the elements of container are iteratively processed by the background worker pool. If more elements are available than workers then sequential processing will occur. In pseudo-code (ignoring parallelisation) the method ara::exec::DeterministicClient::RunWorkerPool behaves as follows:

```
std::array<WorkerThread,4> workers;

template<typename C>
void DeterministicClient::RunWorkerPool( WorkerRunnable<typename C::
    value_type>& w, C& container) noexcept
{
  int count = 0;
  auto c = container.begin();
  while( c != container.end() ) {
    w.Run( *c++, workers[count++] );
    count %= workers.size();
  }
}
```

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. The distribution of the worker threads to processor cores is left to the Operating System.

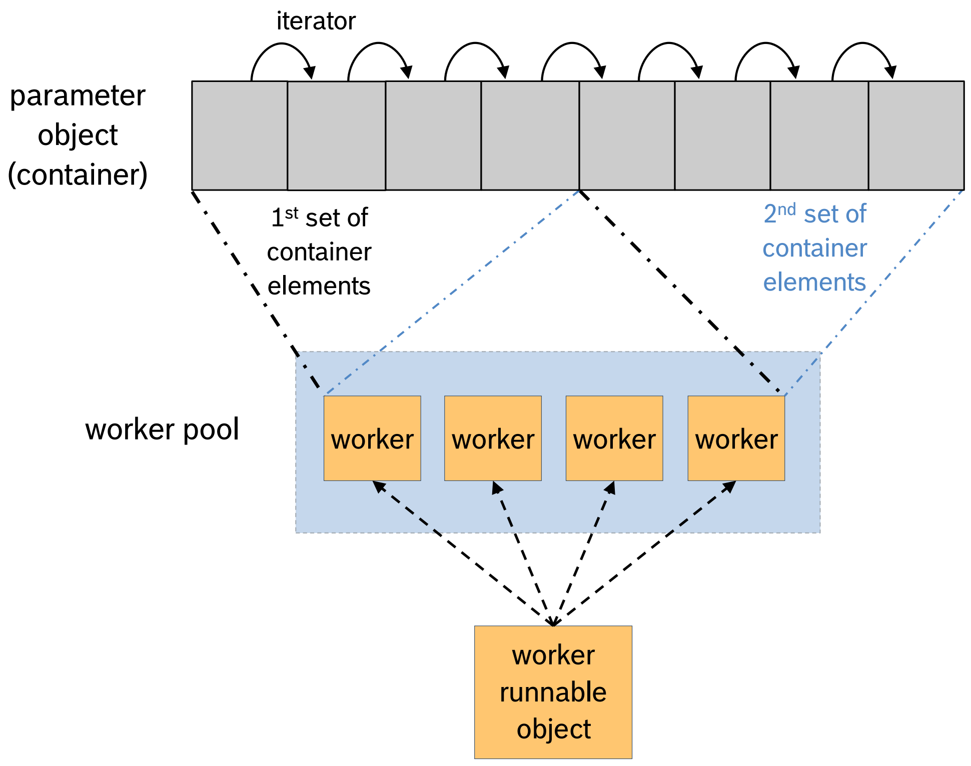
iterator
parameter object (container)
1st set of container elements
2nd set of container elements
worker pool
worker
worker
worker
worker
worker runnable object

**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 `ara::exec::DeterministicClient::WaitForActivation` 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.

#### 7.6.3.3 Random Numbers

**[SWS_EM_01308] Random Numbers** ⌈`Execution Management` shall provide an API `ara::exec::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 `ara::exec::DeterministicClient::RunWorkerPool`, the random numbers are assigned to specific container elements, using the container iterator, to allow deterministic redundant execution.

The `ara::exec::DeterministicClient::SetRandomSeed` API can be used to seed the pseudo random numbers generation to guarantee the deterministic behavior by incorporating it in the DeterministicClient::WaitForNextActivation cycle.

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

#### 7.6.3.4 Time Stamps

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

**[SWS_EM_01310] Get Activation Time** ⌈Execution Management shall provide an API ara::exec::DeterministicClient::GetActivationTime which provides a deterministic timestamp that represents the point in time when the current kRun cycle was activated by ara::exec::DeterministicClient::WaitForActivation (see [SWS_EM_01301]). Deterministic means, that the timestamps are identical for processes which are executed redundantly. Subsequent calls within a cycle shall always return the same value.⌋*(RS_EM_00053, RS_EM_00113)*

**[SWS_EM_01311] Activation Time Unknown** ⌈If ara::exec::DeterministicClient::GetActivationTime is called from outside a kRun cycle, Execution Management shall return kNoTimeStamp.⌋*(RS_EM_00053)*

**[SWS_EM_01312] Get Next Activation Time** ⌈Execution Management shall provide an API ara::exec::DeterministicClient::GetNextActivationTime which provides a deterministic timestamp that represents the point in time when the next kRun cycle will be activated by ara::exec::DeterministicClient::WaitForActivation (see [SWS_EM_01301]). Deterministic means, that the timestamps are identical for processes which are executed redundantly. Subsequent calls within a cycle shall always return the same value.⌋*(RS_EM_00053, RS_EM_00113)*

**[SWS_EM_01313] Next Activation Time Unknown** ⌈In case the next activation time is not known when calling ara::exec::DeterministicClient::GetNextActivationTime, e.g. because of non-equidistant event-triggered activation, Execution Management shall return kNoTimeStamp.⌋*(RS_EM_00053, RS_EM_00113)*

#### 7.6.3.5 Real-Time Resources

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

- Execution Management supports deterministic multithreading to meet high performance demand, see 7.6.3.2
- The integrator needs to assign appropriate resources to the process.
- The integrator needs to assign appropriate scheduling policies. Details and options other than standard POSIX scheduling policies (see [SWS_EM_01014])

heavily depend on the used Operating System, are vendor specific, and are for now out of scope of the Adaptive Platform specification.

- The integrator needs to configure deadline monitoring, possibly execution budget monitoring, and appropriate recovery actions in case of violations. For more details on resources see 7.7.

To make sure that all `processes` which use the `DeterministicClient` APIs get enough computing resources and can finish their cycle in time, it is in particular important to know when the worker pool (`ara::exec::DeterministicClient::RunWorkerPool`) is needed within a `kInit` and `kRun` `ara::exec::DeterministicClient::WaitForActivation` 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 `ara::exec::DeterministicClient::WaitForActivation`-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 reflext the actual number of code instructions, but allows the description of comparative resource needs.

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

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

- *numberOfInstructions* [NormalizedInstructions]

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

- *speedup* = sequental runtime / parallelized runtime

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

- *sequentialInstructionsBegin* [NormalizedInstructions]

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

- *sequentialInstructionsEnd* [NormalizedInstructions]

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

## Examples

### Example 7.5

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

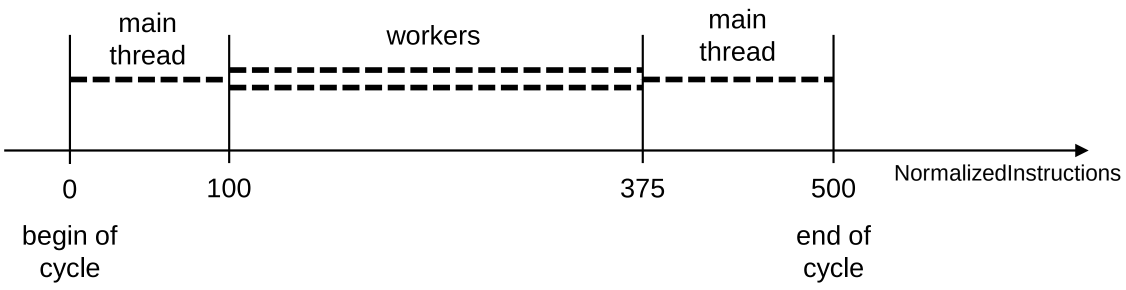

**Figure 7.17: Worker pool used in middle of cycle**

### Example 7.6

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

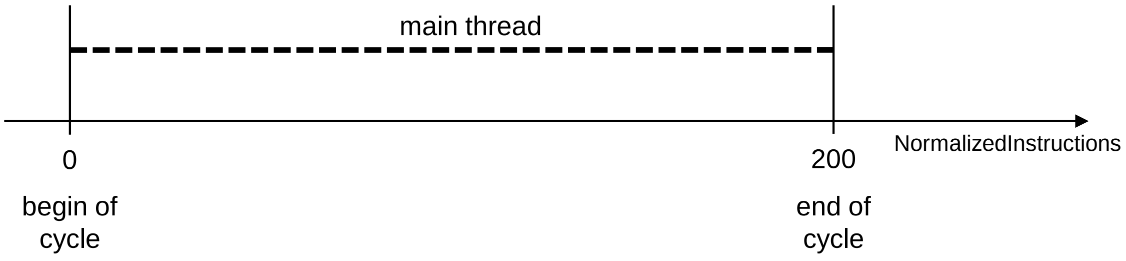

**Figure 7.18: No benefit from worker pool**

**Example 7.7**

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

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

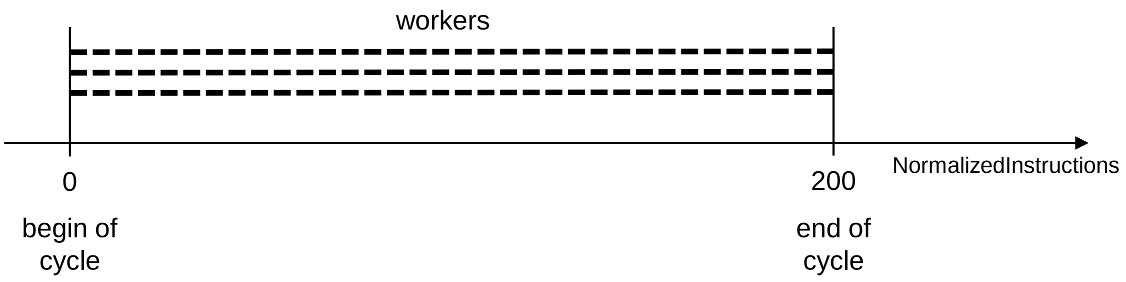

**Figure 7.19: Full utilization of worker pool**

### 7.6.4 Deterministic Synchronization

The API `ara::exec::DeterministicClient::WaitForActivation` is described in 7.6.3 as the wait point in deterministic redundant execution. In this section, more details on synchronization behaviors will be provided for both periodic and event-triggered activation in the execution cycles.

#### 7.6.4.1 DeterministicSyncMaster

A DeterministicSyncMaster is a synchronization control point that receives the synchronization requests through a dedicated communication channel, for example ara::com, and sends the calculated cycle information for the next execution cycle to the connected DeterministicClients in the same domain.

Note that it is not limited to use ara::com or API of other communication channel, and it is up to the vendor to decide which to use. This specification only describes the integration with ara::com API for DeterministicSyncMaster. The integration with other communication APIs is not covered and may be specified in a later release.

Figure 7.20 shows an example of how a DeterministicSyncMaster controls the synchronization for two DeterministicClients of the application process based on ara::com interface for request and response communication.

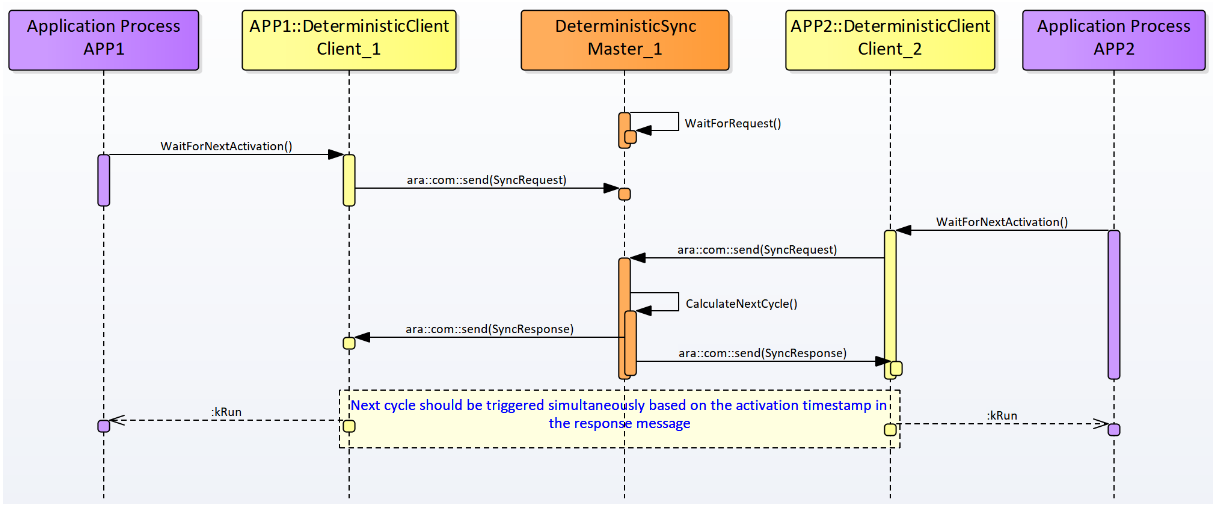
Application Process APP1
APP1::DeterministicClient Client_1
DeterministicSync Master_1
APP2::DeterministicClient Client_2
Application Process APP2
WaitForRequest()
WaitForNextActivation()
ara::com::send(SyncRequest)
WaitForNextActivation()
ara::com::send(SyncRequest)
CalculateNextCycle()
ara::com::send(SyncResponse)
ara::com::send(SyncResponse)
:kRun
Next cycle should be triggered simultaneously based on the activation timestamp in the response message
:kRun

**Figure 7.20: Sequence Diagram of the Synchronization Control Messages with single DeterministicSyncMaster**

For event-triggered activation, a specific policy of the synchronization should be provided. The policy is highly dependent on vendor solution and requirements, for example, the synchronization response is sent to the DeterministicClients only if the synchronization requests from all the processes are received. There can be more complicated policies, e.g. the match of 2 out of 3 synchronization requests are received before the given deadline, which is also known as the M-out-of-N (MooN) policy.

For periodic activation, the DeterministicClients require a single synchronization for the first ara::exec::DeterministicClient::WaitForActivation call that is initiated after the execution of kInit or kServiceDiscovery cycle (see [SWS_EM_01304] for service modification). The activation response includes a global time stamp for the activation of the first kRun cycle, which should also give a reasonable time buffer for receiving the activation response for the DeterministicClients through the channel. All DeterministicClients will count on local time until the activation time is reached and then starts kRun. Further calls of ara::exec::DeterministicClient::WaitForActivation will not send any synchronization request, but just return when the predefined deadline that configured with cycleTimeValue property is reached by the local time counter (see [SWS_EM_01351]). The handling of missing deadline in the kRun cycles for periodic activation is described in [SWS_EM_01352].

For both periodic and event-triggered activation, a set of parameters need to be defined before the DeterministicSyncMaster starts.

**[SWS_EM_01320]**{DRAFT} **Number of DeterministicClients** ⌈The number of DeterministicClients that are connected to the DeterministicSyncMaster shall be set during the initialization of the DeterministicSyncMaster.⌋*(RS_EM_00053)*

**[SWS_EM_01321]**{DRAFT} **Minimum number of required synchronization requests** ⌈The minimum number of required synchronization requests from the connected DeterministicClients in the same domain shall be initialized for DeterministicSyncMaster.⌋*(RS_EM_00053)*

The MooN policy defines a rule for DeterministicSyncMaster to decide when it should response to the synchronization requests for next execution cycle. $N$ is the number of the processes that are connected to the DeterministicSyncMaster in the same domain ([SWS_EM_01320]), and $M$ is the minimum required synchronization requests to be received in the same domain ([SWS_EM_01321]). The usage of MooN can be modified based on the requirements of the redundant execution. For example, when $M$ requests are received ($M < N$), the DeterministicSyncMaster may ignore the rest of the unreceived requests, and start calculating the cycle information for the next activation based on the received requests. The cycle information is encapsulated into a response message and propagated to all of the connected DeterministicClients. If $M$ equals $N$, this means all of the requests from $N$ DeterministicClients should be received before proceeding to the calculation of next cycle .

Note that for the current release only MooN policy is described and the configuration of other policies may be specified in a later release.

**[SWS_EM_01322]**{DRAFT} **Calculation of the next cycle** ⌈DeterministicSyncMaster shall calculate the next activation time based on the MooN policy and the received synchronization request (see [SWS_EM_01325]).⌋*(RS_EM_00053)*

The calculated cycle information is sent via a response message to all connected DeterministicClients (see [SWS_EM_01326]). Based on the response message, a DeterministicClient triggers the next execution cycle on the activation time by returning from the ara::exec::DeterministicClient::WaitForActivation call.

**[SWS_EM_01323]**{DRAFT} **Total kRun loop count** ⌈The total number of the kRun loops shall be set for the target DeterministicClient during the initialization of the DeterministicSyncMaster to indicate when a kTerminate shall be returned from ara::exec::DeterministicClient::WaitForActivation.⌋*(RS_EM_00053)*

**[SWS_EM_01324]**{DRAFT} **Infinite kRun loop** ⌈A setting of kRun loop count with value zero shall indicate an infinite kRun cycle count.⌋*(RS_EM_00053)*

Note: The data type for storing the kRun loop count is implementation specific.

The current specification of DeterministicSyncMaster supports redundant deterministic execution in signal domain and multiple domains. The DeterministicSyncMaster should work as a Time Slave, in order to achieve the global time stamp when it is needed for calculating and distributing the deterministic time stamps of execution cycles. The processes implemented with DeterministicClients should be in the same global time domain as the DeterministicSyncMaster, so they can be synchronized even if they are connected through network or gateway.

For single domain synchronization, both DeterministicClient and DeterministicSyncMaster should use the local time resource for simplicity and efficiency when acquiring the current time. The access of the local time can be achieved by calling ara::core::SteadyClock or POSIX API, e.g. the std::chrono API.

For multiple domain synchronization, both DeterministicClient and DeterministicSyncMaster should be configured to use the same global time resource, for example GPS time. The configuration of the cross network synchronization will be specified in a later release.

The assurance of secure access to the Time Resource should be managed by the Policy Decision Point (PDP) and Policy Enforcement Point (PEP) configurations for Time Slave and Master. For example an Access Manager may be able to grant the permission for DeterministicSyncMaster to access the configured Time Master and Time Resource. As the DeterministicSyncMaster exposes only ara::com and ara::tsync interfaces, access control to functions of the DeterministicSyncMaster should be enforced using IAM for ara::com and ara::tsync.

When the process is running in the execution cycles, each cycle needs to be synchronized by calling ara::exec::DeterministicClient::WaitForActivation. The behaviors to synchronize the process and the redundancies should be performed by the DeterministicSyncMaster, which can be deployed in Execution Management process, Software Lockstep process or in a separate process. Figure 7.21 shows an example of running the DeterministicSyncMaster in a separate process.

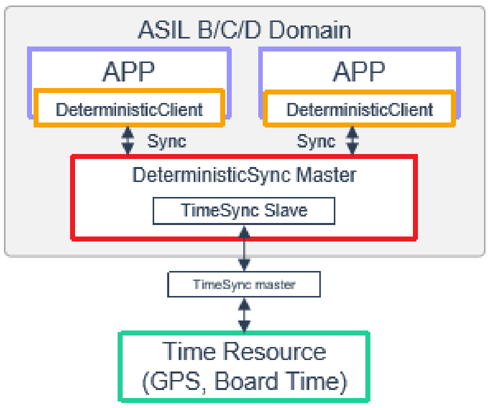

**Figure 7.21: An example deployment of `DeterministicSyncMaster` in a separate process**

The Software Lockstep is an optional framework to ensure identical behavior of the redundantly executed `processes`. The Software Lockstep framework does not necessarily interact with `DeterministicSyncMaster`, but they can be integrated in order to simplify the control logic and reduce the communication effort over `ara::com` or other dedicated communication channels. For example the Software Lockstep may also need to understand the state of each execution cycle, in order to give more reasonable and trusty comparison results. Details of Software Lockstep is out of the scope of the current specification, only the possible integration architectures are briefly discussed. Figure 7.22 gives examples of a Software Lockstep framework in library mode and `process` mode. Details of Software Lockstep will be specified in a later release.

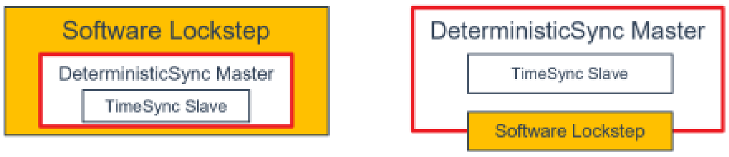

**Figure 7.22: process mode (left) and library mode (right) integration**

For `process` mode of Software Lockstep, the `DeterministicSyncMaster` functionality can be integrated inside the Software Lockstep as a library. For library mode of Software Lockstep, it can be integrated into the `DeterministicSyncMaster process`.

Figure 7.23 and 7.24 illustrates examples of the possible ways to integrate `DeterministicSyncMaster` with the Software Lockstep in `process` mode and library mode for cross domain with two `DeterministicSyncMaster`s. For both modes, the `DeterministicSyncMaster`s should be connected via the synchronization channel for making the final decision. Whether multiple Software Lockstep instances should run in different domains is not within the scope of this concept as it depends on the deployment and solution of the vendor based on the available integration possibilities.

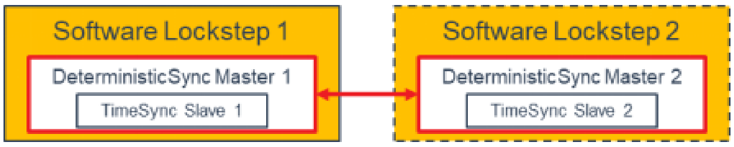

**Figure 7.23: Library Mode with multiple DeterministicSyncMasters**

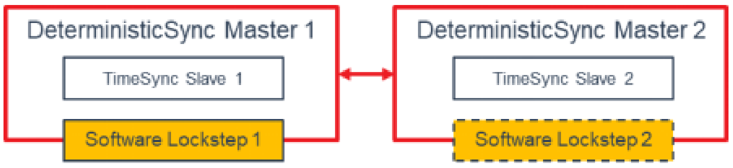

**Figure 7.24: Process Mode with multiple DeterministicSyncMasters**

#### 7.6.4.2 Synchronization Control Messages

In this section, we specify the basic elements of the control messages, in order to run redundant deterministic execution based on platform vendor implementation and data structure. Dedicated interface(s) and data structure(s) will be specified in a later release.

**[SWS_EM_01325]**{DRAFT} **Synchronization Request Message** ⌈The `ara::exec::DeterministicClient::WaitForActivation` for a `DeterministicClient` activation shall send a synchronization request message to the connected `DeterministicSyncMaster`.⌋*(RS_EM_00053)*

A synchronization request should contain at least the following data members:

- **Service ID**: The Service ID of the service skeleton in `DeterministicClient` that the synchronization request was sent from.
- **Instance ID**: The Instance ID of the `process` that sent the synchronization request through the service skeleton.
- **Activation timestamp of the previous cycle**: the activation of the previous cycle is used for calculate the next cycle.
- **Code of the current cycle**: the type of the current cycle is used to determine the type of next execution cycle. Possible codes are `kServiceDiscovery`, `kInit`, `kRun`.
- **Count of the current loop**: the number of the execution loop is used to determine when `ara::exec::DeterministicClient::WaitForActivation` should return `kTerminate`.

The data types of the members are implementation specific, as the interface is between two platform specific elements.

**[SWS_EM_01326]**{DRAFT} **Synchronization Response Message** ⌈A `DeterministicSyncMaster` shall send a synchronization response message to all the connected DeterministicClients when the applied synchronization policy is matched.⌋*(RS_EM_00053)*

A synchronization response should contain at least the following data members:

- **Service ID**: The Service ID of the service skeleton in DeterministicSyncMaster that the synchronization response was sent from.
- **Instance ID**: The Instance ID of the process running the DeterministicSyncMaster that sent the synchronization response through the service skeleton.
- **Activation timestamp for the next cycle**: The calculated activation timestamp of the next execution cycle.
- **Code of the next cycle**: The determined code of the next cycle. Possible values are kRun, kServiceDiscovery, and kTerminate. A kServiceDiscovery code is returned when a service modification is necessary (see [SWS_EM_01304]). The code kTerminate is returned when the total kRun loop count is reached or the termination is requested by Execution Management (see [SWS_EM_01404]).

**[SWS_EM_01327]**{DRAFT} **Return of the wait point API** ⌈A ara::exec::DeterministicClient::WaitForActivation call shall not return until the local time counter reaches the activation timestamp that was sent with the response message of the synchronization for the next kRun cycle.⌋*(RS_EM_00053)*

## 7.7 Resource Limitation

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

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

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

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

### 7.7.1 Resource Configuration

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

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

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

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

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

The following stakeholders are involved in resource management:

- Application Developer

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

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

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

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

- Integrator

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

  - Assignment of resource criticality to Modelled Processes, depending on safety and functional requirements.
  - Depending on the Operating System, maximum resources which cannot be exceeded by design (e.g. Linux cgroups) can be assigned to a process or a group of processes.
  - A scheduling policy has to be applied, so threads of processes with high criticality get guaranteed computing time and finish before a given deadline, while threads of less critical processes might not. For details see section 7.7.3.1.
  - If the summarized maximum RAM needs of all processes, which can be running in parallel at any given time, exceeds the available RAM, this cannot be solved easily by prioritization, since memory assignment to low critical processes cannot just be removed without compromising the process. However, it should be ensured that processes with high criticality have ready access to their maximum resources at any time, while lower criticality processes need to share the remaining resources. For details see 7.7.3.4.

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

### 7.7.2 Resource Monitoring

As far as technically possible, the resources which are actually used by a process should be controlled at any given time. For the entire system, the monitoring part of this activity is fulfilled by the Operating System. For details on CPU time monitoring see 7.7.3.1. For RAM monitoring see 7.7.3.4. The monitoring capabilities depend on the used Operating System. Depending on system requirements and safety goals, an appropriate Operating System has to be chosen and configured accordingly, in combination with other monitoring mechanisms (e.g. for execution deadlines) which are provided by Platform Health Management.

Resource monitoring can serve several purposes, e.g.

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

For processes which are attempting to exceed their configured maximum resource needs (see 7.7.1), one of the following alternatives is valid:

- The resource limit violation or deadline miss is considered a failure and Recovery Actions may need to be initiated. Therefore the specific violation gets reported to the State Management, which then starts Recovery Actions which have been configured beforehand. This will be the standard option for deterministic subsystems (see 7.6.1).

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

### 7.7.3 Application-level Resource Configuration

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

#### 7.7.3.1 CPU Usage

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

#### 7.7.3.2 Core Affinity

**[SWS_EM_02104] 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.

#### 7.7.3.3 Scheduling

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

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

**[SWS_EM_01014] 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] Scheduling FIFO** ⌈Execution Management shall be able to configure FIFO scheduling using policy `SCHED_FIFO`.⌋*(RS_EM_00002)*
- **[SWS_EM_01042] Scheduling Round-Robin** ⌈Execution Management shall be able to configure round-robin scheduling using policy `SCHED_RR`.⌋*(RS_EM_-00002)*
- **[SWS_EM_01043] Scheduling Other** ⌈Execution Management shall be able to configure non real-time scheduling using policy `SCHED_OTHER`.⌋*(RS_EM_-00002)*

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

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] 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].

#### 7.7.3.3.1 Resource Management

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

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

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

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

**[SWS_EM_02106] ResourceGroup assignment** ⌈`Execution Management` shall configure the `process` according to its `ResourceGroup` membership.⌋*(RS_EM_-00005)*

### 7.7.3.4 Memory Budget and Monitoring

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

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

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

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

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

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

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 Modelled 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 Modelled Process may not be loadable anymore.

#### 7.7.3.5 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.

## 7.8 Fault Tolerance

### 7.8.1 Introduction

#### What is Fault-Tolerance?

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

The model adopted for `Execution Management` is outlined in [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.

### 7.8.2 Scope

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

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

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

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

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

### 7.8.3 Threat Model

The main threats which leading to incorrect behavior of software - whether `Application` or `Platform Instance` - is the presence of systematic defects or faults i.e. those incorporated during design phase and remaining dormant untill deployment. Other sources of faults include physical faults, e.g. random hardware failures, that might influence resource allocation and correct execution, and interraction faults which can be a source for incorrect state transition requests.

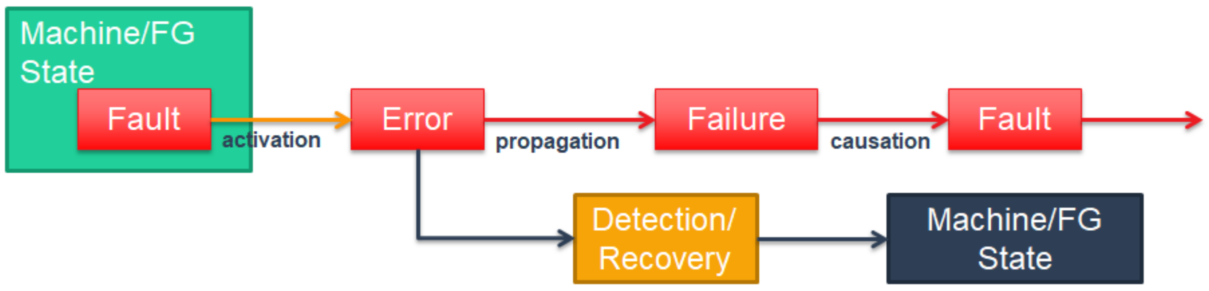

**Figure 7.25: General Fault Tolerance scheme.**

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

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

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

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

### 7.8.4 Execution Management internal Error handling

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

**[SWS_EM_02032]**{DRAFT} **On entry to the Unrecoverable State,** ⌈`Execution Management` shall invoke a pre-cleanup action.⌋*(RS_EM_00150)*

**[SWS_EM_02033]**{DRAFT} **After execution of the pre-cleanup action,** ⌈all `processes` managed by `Execution Management` shall be shutdown.⌋*(RS_EM_00150)*

**[SWS_EM_02034]**{DRAFT} **After all `processes` managed by `Execution Management` terminated,** ⌈a post-cleanup action shall be called.⌋*(RS_EM_00150)*

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

## 7.9 Security

### 7.9.1 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] 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] 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] 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] 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] 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] 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:

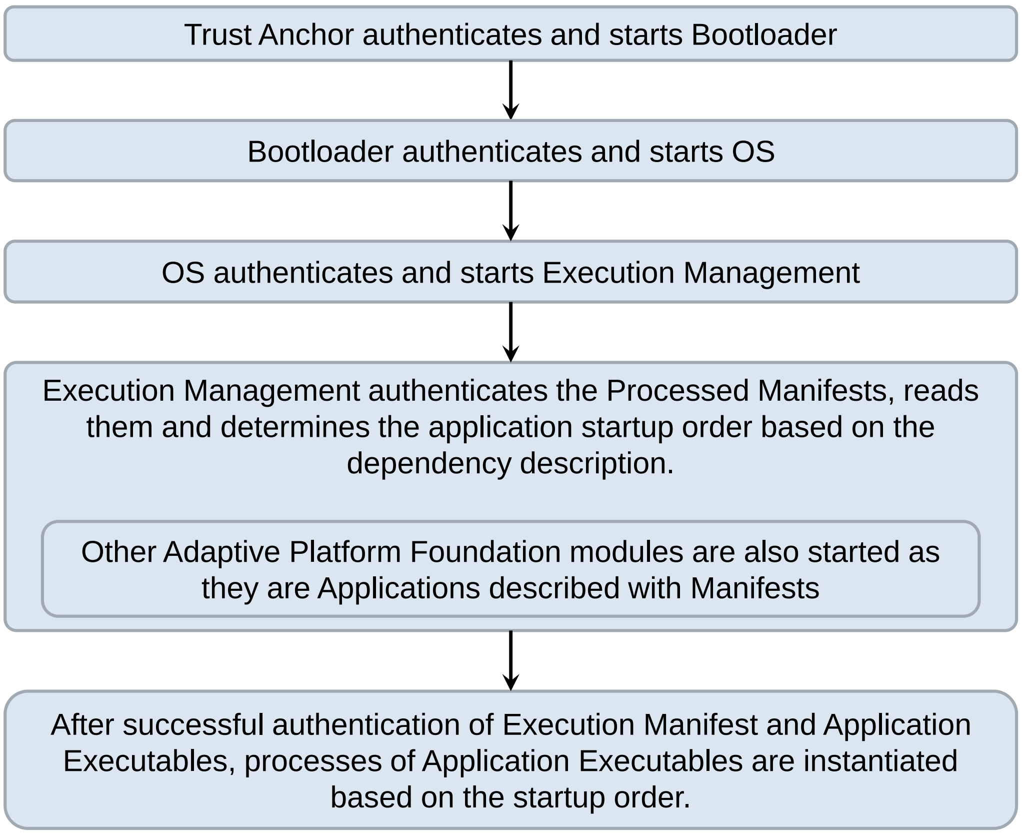
Trust Anchor authenticates and starts Bootloader
Bootloader authenticates and starts OS
OS authenticates and starts Execution Management
Execution Management authenticates the Processed Manifests, reads them and determines the application startup order based on the dependency description.
Other Adaptive Platform Foundation modules are also started as they are Applications described with Manifests
After successful authentication of Execution Manifest and Application Executables, processes of Application Executables are instantiated based on the startup order.

**Figure 7.26: 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.

### 7.9.1.1 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] 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)*

#### 7.9.1.1.1 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.

#### 7.9.1.1.2 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.

### 7.9.2 Identity and Access Management

Following the "Principle of Least Privilege", Identity and Access Management (IAM) was introduced in the Adaptive Platform. IAM allows to assign a minimal set of permissions to access public Functional Cluster Interfaces to `Modelled Processes`. Hence, `Modelled Processes` have to be identifiable during runtime in order to lookup and enforce permissions accordingly.

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

**[SWS_EM_02400]**{DRAFT} **Properties of IAM-configuration assigned to processes** ⌈`Execution Management` shall associate `Modelled Process` identity with `process` during `process` creation.⌋*(RS_EM_00111)*

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

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

# 8 API specification

## 8.1 Type Definitions

### 8.1.1 ExecutionState

**[SWS_EM_02000]** ⌈

| | | |
|---|---|---|
| ***Kind:*** | enumeration | |
| ***Symbol:*** | 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. |
| ***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 7.11, which are an implied states for `Execution Management`. The Initializing state starts when `process` is first scheduled (so no code executed yet) and ends when `kRunning` is reported ([SWS_EM_01004]). The Terminating state starts when termination is requested by `Execution Management` and ends when the `process` terminates ([SWS_EM_01404]). For the reasons mentioned, `Execution Management` assumes that `process` is in initializing state until `kRunning` will be reported by it.

### 8.1.2 ActivationReturnType

**[SWS_EM_02201]** ⌈

| | | |
|---|---|---|
| ***Kind:*** | enumeration | |
| ***Symbol:*** | ActivationReturnType | |
| ***Scope:*** | namespace ara::exec | |
| ***Underlying type:*** | std::uint32_t | |
| ***Syntax:*** | `enum class ActivationReturnType : std::uint32_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]**{OBSOLETE} ⌈

| | | |
|---|---|---|
| ***Kind:*** | enumeration | |
| ***Symbol:*** | 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)*

### 8.1.4 DeterministicClient::TimeStamp

**[SWS_EM_02203]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | type alias |
| ***Symbol:*** | TimeStamp |
| ***Scope:*** | class ara::exec::DeterministicClient |
| ***Derived from:*** | std::chrono::time_point<ara::core::SteadyClock> |
| ***Syntax:*** | `using TimeStamp = std::chrono::time_point<ara::core::SteadyClock>;` |
| ***Header file:*** | #include "ara/exec/deterministic_client.h" |
| ***Description:*** | Time stamp of deterministic cycles . |

⌋*(RS_EM_00052, RS_EM_00053)*

### 8.1.5 ExecutionError

**[SWS_EM_02541]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | type alias |
| ***Symbol:*** | ExecutionError |
| ***Scope:*** | namespace ara::exec |
| ***Derived from:*** | std::uint32_t |
| ***Syntax:*** | `using ExecutionError = std::uint32_t;` |
| ***Header file:*** | #include "ara/exec/execution_error_event.h" |
| ***Description:*** | Represents the execution error. |

⌋*(RS_EM_00101, RS_AP_00122)*

### 8.1.6 ExecutionErrorEvent

**[SWS_EM_02544]**{DRAFT} ⌈

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

⌋*(RS_EM_00101, RS_AP_00116, RS_AP_00122, RS_AP_00124, RS_AP_00140)*

#### 8.1.6.1 ExecutionErrorEvent::executionError

**[SWS_EM_02545]**{DRAFT} ⌈

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

⌋*(RS_EM_00101, RS_AP_00124)*

#### 8.1.6.2 ExecutionErrorEvent::functionGroup

**[SWS_EM_02546]**{DRAFT} ⌈

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

⌋*(RS_EM_00101, RS_AP_00124)*

## 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]** ⌈

| | |
|---|---|
| ***Kind:*** | class |
| ***Symbol:*** | 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]** ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | 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]** ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | ~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]** ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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 |
| | ara::exec::ExecErrc::kCommunication Error | Communication error between Application and Execution Management, e.g. unable to report state for Non-reporting Process. |
| ***Header file:*** | #include "ara/exec/execution_client.h" | |
| ***Description:*** | Interface for a Process to report its internal state to Execution Management. | |

⌋*(RS_EM_00103)*

**[SWS_EM_01403] Reporting Non-reporting Process** ⌈ara::exec::ExecutionClient::ReportExecutionState shall return error kCommunicationError when invoked by a Non-reporting Process.⌋*(RS_EM_00103)*

### 8.2.2 WorkerRunnable class

The WorkerRunnable class provides a base-class defining the expected interface for DeterministicClient worker definition.

**[SWS_EM_02510]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | class | |
| ***Symbol:*** | WorkerRunnable | |
| ***Scope:*** | namespace ara::exec | |
| ***Syntax:*** | `template <typename ValueType>` `class WorkerRunnable {...};` | |
| ***Template param:*** | typename ValueType | Value type of Container passed to Deterministic Client::RunWorkerPool |
| ***Header file:*** | #include "ara/exec/worker_runnable.h" | |
| ***Description:*** | Base-class for implementation of worker runnable for Deterministic Client. | |

⌋*(RS_EM_00052)*

#### 8.2.2.1 WorkerRunnable::Run

**[SWS_EM_02520]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | Run(ValueType &element, ara::exec::WorkerThread &t) | |
| ***Scope:*** | class ara::exec::WorkerRunnable | |
| ***Syntax:*** | `virtual void Run (ValueType &element, ara::exec::WorkerThread &t)=0;` | |
| ***Parameters (in):*** | element | Reference to container element |
| | t | Reference to worker thread (for random numbers) |
| ***Return value:*** | None | |
| ***Header file:*** | #include "ara/exec/worker_runnable.h" | |
| ***Description:*** | Deterministic client worker runnable. | |

⌋*(RS_EM_00052)*

### 8.2.3 WorkerThread class

The WorkerThread class provides class defining the expected interface for DeterministicClient worker threads, in particular, access to deterministic random numbers.

**[SWS_EM_02530]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | class |
| ***Symbol:*** | WorkerThread |
| ***Scope:*** | namespace ara::exec |
| ***Syntax:*** | `class WorkerThread {...};` |
| ***Header file:*** | #include "ara/exec/worker_thread.h" |
| ***Description:*** | Class to implement worker thread for Deterministic Client . |

⌋*(RS_EM_00052)*

#### 8.2.3.1 WorkerThread::WorkerThread

**[SWS_EM_02531]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | WorkerThread() |
| ***Scope:*** | class ara::exec::WorkerThread |
| ***Syntax:*** | `WorkerThread ();` |
| ***Header file:*** | #include "ara/exec/worker_thread.h" |
| ***Description:*** | Constructor . |

⌋*(RS_EM_00052)*

#### 8.2.3.2 WorkerThread::~WorkerThread

**[SWS_EM_02532]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | ~WorkerThread() |
| ***Scope:*** | class ara::exec::WorkerThread |
| ***Syntax:*** | `virtual ~WorkerThread ();` |
| ***Header file:*** | #include "ara/exec/worker_thread.h" |
| ***Description:*** | Destructor . |

⌋*(RS_EM_00052)*

#### 8.2.3.3 WorkerThread::GetRandom

**[SWS_EM_02540]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | GetRandom() | |
| ***Scope:*** | class ara::exec::WorkerThread | |
| ***Syntax:*** | `uint64_t GetRandom () noexcept;` | |
| ***Return value:*** | uint64_t | Deterministic random number |
| ***Exception Safety:*** | noexcept | |
| ***Header file:*** | #include "ara/exec/worker_thread.h" | |
| ***Description:*** | Returns a deterministic pseudo-random number which is unique for each container element. | |

⌋*(RS_EM_00052)*

### 8.2.4 DeterministicClient class

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

**[SWS_EM_02210]** ⌈

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

⌋*(RS_EM_00052)*

#### 8.2.4.1 DeterministicClient::DeterministicClient

**[SWS_EM_02211]** ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | DeterministicClient() |
| ***Scope:*** | class ara::exec::DeterministicClient |
| ***Syntax:*** | `DeterministicClient () 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)*

### 8.2.4.2 DeterministicClient::~DeterministicClient

**[SWS_EM_02215]** ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | ~DeterministicClient() |
| ***Scope:*** | class ara::exec::DeterministicClient |
| ***Syntax:*** | `~DeterministicClient () 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)*

### 8.2.4.3 DeterministicClient::WaitForNextActivation

**[SWS_EM_02216]**{OBSOLETE} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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)*

### 8.2.4.4 DeterministicClient::WaitForActivation

**[SWS_EM_02217]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | WaitForActivation() | |
| ***Scope:*** | class ara::exec::DeterministicClient | |
| ***Syntax:*** | `ara::core::Result<ActivationReturnType> WaitForActivation () noexcept;` | |
| ***Return value:*** | ara::core::Result< ActivationReturn Type > | Process control value (or error) |
| ***Exception Safety:*** | noexcept | |
| ***Errors:*** | ara::exec::ExecErrc::kRegisterServices | Process registers its communication services. |

▽

△

| | | |
|---|---|---|
| | ara::exec::ExecErrc::kService Discovery | Process performs communication service discovery. |
| | ara::exec::ExecErrc::kInit | Process initializes its internal data structures. |
| | ara::exec::ExecErrc::kRun | Process performs one cycle of its normal cyclic execution. |
| | ara::exec::ExecErrc::kTerminate | Process prepares to terminate. |
| ***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)*

#### 8.2.4.5 DeterministicClient::RunWorkerPool

**[SWS_EM_02220]**{OBSOLETE} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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)*

#### 8.2.4.6 DeterministicClient::RunWorkerPool

**[SWS_EM_02221]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | RunWorkerPool(WorkerRunnable< ValueType > &runnableObj, Container &container) |
| ***Scope:*** | class ara::exec::DeterministicClient |

▽

△

| | | |
|---|---|---|
| ***Syntax:*** | `template <typename ValueType, typename Container> void RunWorkerPool (WorkerRunnable< ValueType > &runnableObj, Container &container) noexcept;` | |
| ***Template param:*** | ValueType | Element type of container |
| | Container | Container for which method WorkerRunnable::Run is invoked for each element |
| ***Parameters (in):*** | runnableObj | Object derived from WorkerRunnable that provides a method called Run(...), which will be called on every container element |
| | container | C++ container which supports a standard iterator interface with - begin() - end() - operator*() operator++ |
| ***Return value:*** | None | |
| ***Exception Safety:*** | noexcept | |
| ***Header file:*** | #include "ara/exec/deterministic_client.h" | |
| ***Description:*** | Run a deterministic worker pool.<br><br>Uses a pool of Deterministic workers to call a method WorkerRunnable::Run for every element of the container. The sequential iteration is guaranteed by using the container's increment operator. The API provides the guarantee that no other iteration scheme is used.<br><br>This function shall not participate in overload resolution unless unless ValueType is compatible with Container::value_type. | |

⌋*(RS_EM_00053)*

#### 8.2.4.7 DeterministicClient::GetRandom

**[SWS_EM_02225]** ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | GetRandom() | |
| ***Scope:*** | class ara::exec::DeterministicClient | |
| ***Syntax:*** | `uint64_t GetRandom () 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:*** | Return deterministic sequence of random numbers.<br><br>This returns the next in a sequence of 'Deterministic' random numbers. Deterministic' means, that the returned random numbers are identical within redundant DeterministicClient::WaitFor NextActivation() cycles, which are used within redundantly executed Processes. | |

⌋*(RS_EM_00053)*

#### 8.2.4.8 DeterministicClient::SetRandomSeed

**[SWS_EM_02226]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | SetRandomSeed(uint64_t seed) | |
| ***Scope:*** | class ara::exec::DeterministicClient | |
| ***Syntax:*** | `void SetRandomSeed (uint64_t seed) noexcept;` | |
| ***DIRECTION NOT DEFINED*** | seed | – |
| ***Return value:*** | None | |
| ***Exception Safety:*** | noexcept | |
| ***Header file:*** | #include "ara/exec/deterministic_client.h" | |
| ***Description:*** | Seed random number generator used for redundantly executed deterministic clients. | |

⌋*(RS_EM_00053)*

### 8.2.4.9 DeterministicClient::GetActivationTime

**[SWS_EM_02230]**{OBSOLETE} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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)*

### 8.2.4.10 DeterministicClient::GetNextActivationTime

**[SWS_EM_02235]**{OBSOLETE} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | 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)*

### 8.2.4.11 DeterministicClient::GetActivationTime

**[SWS_EM_02231]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | GetActivationTime() | |
| ***Scope:*** | class ara::exec::DeterministicClient | |
| ***Syntax:*** | `ara::core::Result<TimeStamp> GetActivationTime () noexcept;` | |
| ***Return value:*** | ara::core::Result< TimeStamp > | TimeStamp of current activation cycle |
| ***Exception Safety:*** | noexcept | |
| ***Errors:*** | ara::exec::ExecErrc::kNoTimeStamp | Time stamp not available |
| ***Header file:*** | #include "ara/exec/deterministic_client.h" | |
| ***Description:*** | TimeStamp of activation point.<br>This method 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)*

### 8.2.4.12 DeterministicClient::GetNextActivationTime

**[SWS_EM_02236]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | GetNextActivationTime() | |
| ***Scope:*** | class ara::exec::DeterministicClient | |
| ***Syntax:*** | `ara::core::Result<TimeStamp> GetNextActivationTime () noexcept;` | |
| ***Return value:*** | ara::core::Result< TimeStamp > | TimeStamp of next activation cycle |
| ***Exception Safety:*** | noexcept | |

▽

△

| | | |
|---|---|---|
| ***Errors:*** | ara::exec::ExecErrc::kNoTimeStamp | Time stamp not available |
| ***Header file:*** | #include "ara/exec/deterministic_client.h" | |
| ***Description:*** | Timestamp of next activation point.<br><br>This method provides the timestamp that represents the point in time when the next activation will be triggered by DeterministicClient::WaitForNextActivation() with return value kRun. Subsequent calls within an activation cycle will always provide the same value. The same value will also be provided within redundantly executed Process | |

⌋*(RS_EM_00053)*

### 8.2.5 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:*** | FunctionGroup |
| ***Scope:*** | namespace ara::exec |
| ***Syntax:*** | `class FunctionGroup final {...};` |
| ***Header file:*** | #include "ara/exec/function_group.h" |
| ***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)*

#### 8.2.5.1 FunctionGroup::Preconstruct

**[SWS_EM_02264]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | Preconstruct(ara::core::StringView metaModelIdentifier) | |
| ***Scope:*** | class ara::exec::FunctionGroup | |
| ***Syntax:*** | `static Result<FunctionGroup::CtorToken> Preconstruct (ara::core::StringView metaModelIdentifier) noexcept;` | |
| ***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:*** | #include "ara/exec/function_group.h" | |
| ***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)*

### 8.2.5.2 FunctionGroup::FunctionGroup

**[SWS_EM_02265]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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/function_group.h" | |
| ***Description:*** | Constructor that creates FunctionGroup instance. | |
| ***Notes:*** | Please note that token is destructed during object construction! | |

⌋*(RS_EM_00101)*

### 8.2.5.3 FunctionGroup::~FunctionGroup

**[SWS_EM_02266]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | ~FunctionGroup() |
| ***Scope:*** | class ara::exec::FunctionGroup |
| ***Syntax:*** | `~FunctionGroup () noexcept;` |
| ***Exception Safety:*** | noexcept |
| ***Header file:*** | #include "ara/exec/function_group.h" |

▽

△

| | |
|---|---|
| ***Description:*** | Destructor of the FunctionGroup instance. |

⌋(*RS_EM_00101*)

#### 8.2.5.4 FunctionGroup::operator==

**[SWS_EM_02267]**{DRAFT} ⌈

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

⌋(*RS_EM_00101*)

#### 8.2.5.5 FunctionGroup::operator!=

**[SWS_EM_02268]**{DRAFT} ⌈

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

⌋(*RS_EM_00101*)

### 8.2.6 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:*** | FunctionGroupState |
| ***Scope:*** | namespace ara::exec |
| ***Syntax:*** | `class FunctionGroupState final {...};` |
| ***Header file:*** | #include "ara/exec/function_group_state.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)*

#### 8.2.6.1 FunctionGroupState::Preconstruct

**[SWS_EM_02270]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | Preconstruct(const FunctionGroup &functionGroup, ara::core::StringView metaModelIdentifier) | |
| ***Scope:*** | class ara::exec::FunctionGroupState | |
| ***Syntax:*** | `static ara::core::Result<FunctionGroupState::CtorToken> Preconstruct (const FunctionGroup &functionGroup, ara::core::StringView metaModel Identifier) 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/function_group_state.h" | |
| ***Description:*** | Pre construction method for FunctionGroupState.<br><br>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)*

### 8.2.6.2 FunctionGroupState::FunctionGroupState

**[SWS_EM_02271]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | 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/function_group_state.h" | |
| ***Description:*** | Constructor that creates FunctionGroupState instance. | |
| ***Notes:*** | Please note that token is destructed during object construction! | |

⌋*(RS_EM_00101)*

### 8.2.6.3 FunctionGroupState::~FunctionGroupState

**[SWS_EM_02272]**{DRAFT} ⌈

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

⌋*(RS_EM_00101)*

### 8.2.6.4 FunctionGroupState::operator==

**[SWS_EM_02273]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | operator==(const FunctionGroupState &other) | |
| ***Scope:*** | class ara::exec::FunctionGroupState | |
| ***Syntax:*** | `bool operator== (const FunctionGroupState &other) const noexcept;` | |
| ***Parameters (in):*** | other | FunctionGroupState instance to compare this one with. |

▽

△

| | | |
|---|---|---|
| ***Return value:*** | bool | true in case both FunctionGroupStates are representing exactly the same meta-model element, false otherwise. |
| ***Exception Safety:*** | noexcept | |
| ***Thread Safety:*** | Thread-safe | |
| ***Header file:*** | #include "ara/exec/function_group_state.h" | |
| ***Description:*** | eq operator to compare with other FunctionGroupState instance. | |

⌋*(RS_EM_00101)*

#### 8.2.6.5 FunctionGroupState::operator!=

**[SWS_EM_02274]**{DRAFT} ⌈

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

⌋*(RS_EM_00101)*

### 8.2.7 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:*** | StateClient |
| ***Scope:*** | namespace ara::exec |
| ***Syntax:*** | `class StateClient final {...};` |
| ***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)*

#### 8.2.7.1 StateClient::StateClient

**[SWS_EM_02276]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | StateClient(std::function< void(ara::exec::FunctionGroup &)> undefinedStateCallback) | |
| ***Scope:*** | class ara::exec::StateClient | |
| ***Syntax:*** | `StateClient (std::function< void(ara::exec::FunctionGroup &)> undefinedStateCallback) noexcept;` | |
| ***Parameters (inout):*** | undefinedStateCallback | callback to be invoked by StateClient library if a FunctionGroup changes its state unexpectedly to an Undefined Function Group State, i.e. without previous request by SetState(). The affected Function Group is provided as an argument to the callback. |
| ***Exception Safety:*** | noexcept | |
| ***Header file:*** | #include "ara/exec/state_client.h" | |
| ***Description:*** | Constructor that creates State Client instance.<br>Registers given callback which is called in case a Function Group changes its state unexpectedly to an Undefined Function Group State. | |

⌋*(RS_EM_00101, RS_AP_00120, RS_AP_00121, RS_AP_00132)*

#### 8.2.7.2 StateClient::~StateClient

**[SWS_EM_02277]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | ~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)*

### 8.2.7.3 StateClient::SetState

**[SWS_EM_02278]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | SetState(const FunctionGroupState &state) | |
| ***Scope:*** | class ara::exec::StateClient | |
| ***Syntax:*** | `ara::core::Future<void> SetState (const FunctionGroupState &state) const noexcept;` | |
| ***Parameters (in):*** | state | representing meta-model definition of a state inside a specific Function Group. Execution Management will perform state transition from the current state to the state identified by this parameter. |
| ***Return value:*** | ara::core::Future< void > | void if requested transition is successful, otherwise it returns 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::kFailed UnexpectedTerminationOnExit | if Unexpected Termination in Process of previous Function Group State happened. |
| | ara::exec::ExecErrc::kFailed UnexpectedTerminationOnEnter | if Unexpected Termination in Process of target Function Group State happened. |
| | ara::exec::ExecErrc::kInvalidArguments | if arguments passed doesn't appear to be valid (e.g. after a software update, given functionGroup doesn't exist anymore) |
| | ara::exec::ExecErrc::kCommunication Error | if StateClient can't communicate with Execution Management (e.g. IPC link is down) |
| | ara::exec::ExecErrc::kAlreadyInState | if the FunctionGroup is already in the requested state |
| | ara::exec::ExecErrc::kInTransitionTo SameState | if a transition to the requested state is already ongoing |
| | ara::exec::ExecErrc::kGeneralError | if any other error occurs. |
| ***Header file:*** | #include "ara/exec/state_client.h" | |
| ***Description:*** | Method to request state transition for a single Function Group.<br>This method will request Execution Management to perform state transition and return immediately. Returned ara::core::Future can be used to determine result of requested transition. | |

⌋*(RS_EM_00101)*

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

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

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 ara::exec::StateClient::SetState requests will be issued by mistake. It is in the best interest of Execution Management to inform requester (instance of ara::exec::StateClient) of the ongoing transition, that it had been canceled by a newer request as soon as possible.

**[SWS_EM_02298]**{DRAFT} **Canceling ongoing state transition** ⌈When Execution Management receives ara::exec::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 ara::exec::StateClient) about cancellation, before informing new requester that the new request has been accepted. Both requesters could be the same instance of ara::exec::StateClient. There are no other requirements or assumtions on order in which requests from ara::exec::StateClient::SetState are processed.

#### 8.2.7.4 StateClient::GetInitialMachineStateTransitionResult

**[SWS_EM_02279]**{DRAFT} ⌈

| | |
|---|---|
| ***Kind:*** | function |
| ***Symbol:*** | 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 `ara::exec::StateClient::SetState` apply for `ara::exec::StateClient::GetInitialMachineStateTransitionResult`.

#### 8.2.7.5 StateClient::GetExecutionError

**[SWS_EM_02542]**{DRAFT} ⌈

| | | |
|---|---|---|
| ***Kind:*** | function | |
| ***Symbol:*** | GetExecutionError(const ara::exec::FunctionGroup &functionGroup) | |
| ***Scope:*** | class ara::exec::StateClient | |
| ***Syntax:*** | `ara::core::Result<ara::exec::ExecutionErrorEvent> GetExecutionError (const ara::exec::FunctionGroup &functionGroup) noexcept;` | |
| ***Parameters (in):*** | functionGroup | Function Group of interest. |
| ***Return value:*** | ara::core::Result< ara::exec::Execution ErrorEvent > | The execution error which changed the given Function Group to an Undefined Function Group State. |
| ***Exception Safety:*** | noexcept | |
| ***Thread Safety:*** | thread-safe | |
| ***Errors:*** | ara::exec::ExecErrc::kFailed | Given Function Group is not in an Undefined Function Group State. |
| ***Header file:*** | #include "ara/exec/state_client.h" | |

▽

△

| | |
|---|---|
| ***Description:*** | Returns the execution error which changed the given Function Group to an Undefined Function Group State.<br><br>This function will return with error and will not return an ExecutionErrorEvent object, if the given Function Group is in a defined Function Group state again. |

⌋*(RS_EM_00101, RS_AP_00120, RS_AP_00121, RS_AP_00132, RS_AP_00128)*

**[SWS_EM_02543]**{DRAFT} **Default value for ExecutionError** ⌈In case of `Unexpected Termination` or `Unexpected Self-termination` of a `Modelled Process` which does not have an `executionError` configured, `Execution Management` shall report the `ExecutionError` value 1.⌋*(RS_EM_00101)*

## 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:*** | ExecErrc | |
| ***Scope:*** | namespace ara::exec | |
| ***Underlying type:*** | ara::core::ErrorDomain::CodeType | |
| ***Syntax:*** | `enum class ExecErrc : ara::core::ErrorDomain::CodeType {...};` | |
| ***Values:*** | kGeneralError= 1 | Some unspecified error occurred |
| | kInvalidArguments= 2 | Invalid argument was passed |
| | kCommunicationError= 3 | Communication error occurred |
| | kMetaModelError= 4 | Wrong meta model identifier passed to a function |
| | kCancelled= 5 | Transition to the requested Function Group state was cancelled by a newer request |
| | kFailed= 6 | Requested operation could not be performed |
| | kFailedUnexpectedTerminationOnExit= 7 | Unexpected Termination during transition in Process of previous Function Group State happened |
| | kFailedUnexpectedTerminationOnEnter= 8 | Unexpected Termination during transition in Process of target Function Group State happened |
| | kInvalidTransition= 9 | Transition invalid (e.g. to Terminating when already in Terminating state) |
| | kAlreadyInState= 10 | Transition to the requested Function Group state failed because it is already in requested state |
| | kInTransitionToSameState= 11 | Transition to the requested Function Group state failed because transition to requested state is already in progress |
| | kNoTimeStamp= 12 | DeterministicClient time stamp information is not available |
| ***Header file:*** | #include "ara/exec/exec_error_domain.h" | |
| ***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:*** | ExecException |
| ***Scope:*** | namespace ara::exec |
| ***Base class:*** | ara::core::Exception |
| ***Syntax:*** | `class ExecException : public Exception {...};` |
| ***Header file:*** | #include "ara/exec/exec_error_domain.h" |
| ***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:*** | 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:*** | GetExecErrorDomain() | |
| ***Scope:*** | namespace ara::exec | |
| ***Syntax:*** | `const ara::core::ErrorDomain& GetExecErrorDomain () noexcept;` | |
| ***Return value:*** | const ara::core::ErrorDomain & | 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:*** | 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:*** | ExecErrorDomain |
| ***Scope:*** | namespace ara::exec |
| ***Base class:*** | ara::core::ErrorDomain |
| ***Syntax:*** | `class ExecErrorDomain final : public ErrorDomain {...};` |
| ***Unique ID:*** | 0x8000'0000'0000'0202 |
| ***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:*** | ExecErrorDomain() | |
| ***Scope:*** | class ara::exec::ExecErrorDomain | |
| ***Syntax:*** | `ExecErrorDomain () noexcept;` | |
| ***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:*** | Name() | |
| ***Scope:*** | class ara::exec::ExecErrorDomain | |
| ***Syntax:*** | `const char* Name () const noexcept override;` | |
| ***Return value:*** | const char * | "Exec". |
| ***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:*** | Message(CodeType errorCode) | |
| ***Scope:*** | class ara::exec::ExecErrorDomain | |
| ***Syntax:*** | `const char* Message (CodeType errorCode) const noexcept override;` | |
| ***Parameters (in):*** | errorCode | The error code number. |
| ***Return value:*** | const char * | The message associated with the error code. |
| ***Exception Safety:*** | 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:*** | ThrowAsException(const ara::core::ErrorCode &errorCode) | |
| ***Scope:*** | class ara::exec::ExecErrorDomain | |
| ***Syntax:*** | `void ThrowAsException (const ara::core::ErrorCode &errorCode) const noexcept(false) override;` | |
| ***Parameters (in):*** | errorCode | The error to throw. |
| ***Return value:*** | None | |
| ***Exception Safety:*** | 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.

| ***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.<br>**Tags:**<br>atp.Status=draft<br>atp.recommendedPackage=DeterministicClients | | | |
| ***Base*** | *ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadablePackageElement* | | | |
| ***Attribute*** | ***Type*** | ***Mult.*** | ***Kind*** | ***Note*** |
| cycleTimeValue | TimeValue | 0..1 | attr | This attribute represents the cycle time for execution of a DeterministicClient activation cycle. |
| numberOf Workers | PositiveInteger | 0..1 | attr | Number of independent workers that process data-sets. Size of the worker pool shall be decided based on availability of resources like processor cores or memory. |

**Table A.1: DeterministicClient**

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

**Table A.2: 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.<br><br>**Tags:**atp.Status=draft | | | |
| *Base* | *ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable* | | | |
| *Attribute* | *Type* | *Mult.* | *Kind* | *Note* |
| hardware Platform | String | 0..1 | attr | This attribute represents a textual identification of the target platform. |
| initResource | DeterministicClient Resource | 0..1 | aggr | This represents the computing resource needs of a DeterministicClient::WaitForNextActivation kInit cycle.<br><br>**Tags:**atp.Status=draft |
| runResource | DeterministicClient Resource | 0..1 | aggr | This represents the computing resource needs of a DeterministicClient::WaitForNextActivation kRun cycle.<br><br>**Tags:**atp.Status=draft |

**Table A.3: DeterministicClientResourceNeeds**

| *Class* | **Executable** | | | |
|---|---|---|---|---|
| *Package* | M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure | | | |
| *Note* | This meta-class represents an executable program.<br><br>**Tags:**<br>atp.Status=draft<br>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.<br><br>**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.<br><br>**Tags:**atp.Status=draft |
| version | StrongRevisionLabel String | 0..1 | attr | Version of the executable.<br><br>**Tags:**atp.Status=draft |

**Table A.4: 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<br><br>**Tags:**atp.Status=draft |
| ***Literal*** | ***Description*** |
| doesNotReport ExecutionState | The Executable shall not report its execution state to the Execution Management.<br><br>**Tags:**atp.EnumerationLiteralIndex=1 |
| reportsExecution State | The Executable shall report its execution state to the Execution Management.<br><br>**Tags:**atp.EnumerationLiteralIndex=0 |

**Table A.5: ExecutionStateReportingBehaviorEnum**

| ***Class*** | **Machine** | | | |
|---|---|---|---|---|
| ***Package*** | M2::AUTOSARTemplates::AdaptivePlatform::MachineManifest | | | |
| ***Note*** | Machine that represents an Adaptive Autosar Software Stack.<br><br>**Tags:**<br>atp.Status=draft<br>atp.recommendedPackage=Machines | | | |
| ***Base*** | *ARElement*, *ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable* | | | |
| ***Attribute*** | ***Type*** | ***Mult.*** | ***Kind*** | ***Note*** |
| default Application Timeout | EnterExitTimeout | 0..1 | aggr | This aggration defines a default timeout in the context of a given Machine with respect to the launching and termination of applications.<br><br>**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.<br><br>**Stereotypes:** atpSplitable<br>**Tags:**<br>atp.Splitkey=environmentVariable, environment Variable.variationPoint.shortLabel<br>atp.Status=draft |
| machineDesign | MachineDesign | 1 | ref | Reference to the MachineDesign this Machine is implementing.<br><br>**Tags:**atp.Status=draft |
| module Instantiation | AdaptiveModule Instantiation | * | aggr | Configuration of Adaptive Autosar module instances that are running on the machine.<br><br>**Stereotypes:** atpSplitable<br>**Tags:**<br>atp.Splitkey=moduleInstantiation.shortName<br>atp.Status=draft |
| processor | Processor | 1..* | aggr | This represents the collection of processors owned by the enclosing machine.<br><br>**Tags:**atp.Status=draft |

▽

△

| *Class* | **Machine** | | | |
|---|---|---|---|---|
| secure Communication Deployment | SecureCommunication Deployment | * | aggr | Deployment of secure communication protocol configuration settings to crypto module entities.<br><br>**Stereotypes:** atpSplitable<br>**Tags:**<br>atp.Splitkey=secureCommunicationDeployment.short Name<br>atp.Status=draft |
| 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.6: Machine**

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

**Table A.7: ModeDeclaration**

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

**Table A.8: NormalizedInstruction**

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

▽

△

| ***Class*** | **Process** | | | |
|---|---|---|---|---|
| design | ProcessDesign | 0..1 | ref | This reference represents the identification of the design-time representation for the Process that owns the reference.<br><br>**Tags:**atp.Status=draft |
| deterministic Client | DeterministicClient | 0..1 | ref | This reference adds further execution characteristics for deterministic clients.<br><br>**Tags:**atp.Status=draft |
| executable | Executable | 0..1 | ref | Reference to executable that is executed in the process.<br><br>**Stereotypes:** atpUriDef<br>**Tags:**atp.Status=draft |
| functionCluster Affiliation | String | 0..1 | attr | This attribute specifies which functional cluster the process is affiliated with. |
| numberOf RestartAttempts | PositiveInteger | 0..1 | attr | This attribute defines how often a process shall be restarted if the start fails.<br><br>numberOfRestartAttempts = "0" OR Attribute not existing, start once<br><br>numberOfRestartAttempts = "1", start a second time |
| 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.<br><br>**Tags:**atp.Status=draft |
| securityEvent | SecurityEventDefinition | * | ref | The reference identifies the collection of SecurityEvents that can be reported by the enclosing SoftwareCluster.<br><br>**Stereotypes:** atpSplitable; atpUriDef<br>**Tags:**<br>atp.Splitkey=securityEvent<br>atp.Status=draft |
| stateDependent StartupConfig | StateDependentStartup Config | * | aggr | Applicable startup configurations.<br><br>**Tags:**atp.Status=draft |

**Table A.9: Process**

| ***Class*** | **ProcessArgument** | | | |
|---|---|---|---|---|
| ***Package*** | M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest | | | |
| ***Note*** | This meta-class has the ability to define command line arguments for processing by the Main function.<br><br>**Tags:**atp.Status=draft | | | |
| ***Base*** | *ARObject* | | | |
| ***Attribute*** | ***Type*** | ***Mult.*** | ***Kind*** | ***Note*** |
| argument | String | 0..1 | attr | This represents one command-line argument to be processed by the executable software. |

**Table A.10: ProcessArgument**

| ***Class*** | **ResourceGroup** | | | |
|---|---|---|---|---|
| ***Package*** | M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::AdaptiveModule Implementation | | | |
| ***Note*** | This meta-class represents a resource group that limits the resource usage of a collection of processes.<br><br>**Tags:**atp.Status=draft | | | |
| ***Base*** | *ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable* | | | |
| ***Attribute*** | ***Type*** | ***Mult.*** | ***Kind*** | ***Note*** |
| cpuUsage | PositiveInteger | 0..1 | attr | CPU resource limit in percentage of the total CPU capacity on the machine. |
| memUsage | PositiveInteger | 0..1 | attr | Memory limit in bytes. |

**Table A.11: ResourceGroup**

| ***Class*** | **StartupConfig** | | | |
|---|---|---|---|---|
| ***Package*** | M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest | | | |
| ***Note*** | This meta-class represents a reusable startup configuration for processes..<br><br>**Tags:**atp.Status=draft | | | |
| ***Base*** | *ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable* | | | |
| ***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.<br><br>**Tags:**atp.Status=draft |
| executionError | ProcessExecutionError | 0..1 | ref | this reference is used to identify the applicable execution error<br><br>**Tags:**atp.Status=draft |
| process Argument | ProcessArgument | * | aggr | This aggregation represents the collection of command-line arguments applicable to the enclosing StartupConfig.<br><br>**Tags:**atp.Status=draft |
| scheduling Policy | String | 0..1 | attr | This attribute represents the ability to define the scheduling policy for the initial thread of the application. |
| scheduling Priority | Integer | 0..1 | attr | This is the scheduling priority requested by the application itself. |
| termination Behavior | TerminationBehavior Enum | 0..1 | attr | This attribute defines the termination behavior of the Process. |
| timeout | EnterExitTimeout | 0..1 | aggr | This aggregation can be used to specify the timeouts for launching and terminating the process depending on the StartupConfig.<br><br>**Tags:**atp.Status=draft |

**Table A.12: StartupConfig**

| ***Class*** | **StateDependentStartupConfig** |
|---|---|
| ***Package*** | M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest |
| ***Note*** | This meta-class defines the startup configuration for the process depending on a collection of machine states.<br><br>**Tags:**atp.Status=draft |

▽

△

| *Class* | **StateDependentStartupConfig** | | | |
|---|---|---|---|---|
| *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.<br>**Tags:**atp.Status=draft |
| functionGroup State | ModeDeclaration | * | iref | This represent the applicable functionGroupMode.<br>**Tags:**atp.Status=draft<br>**InstanceRef implemented by:**FunctionGroupStateIn FunctionGroupSetInstanceRef |
| resource Consumption | ResourceConsumption | 0..1 | aggr | This aggregation provides the ability to define resource consumption boundaries on a per-process-startup-config basis.<br>**Tags:**atp.Status=draft |
| resourceGroup | ResourceGroup | 1 | ref | Reference to an applicable resource group.<br>**Tags:**atp.Status=draft |
| startupConfig | StartupConfig | 1 | ref | Reference to a reusable startup configuration with startup parameters.<br>**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. | | | |
| *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**

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

**Table A.15: TerminationBehaviorEnum**

# B 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.

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

### B.1.1 Added Traceables in 17-10

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

▽

△

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

### B.1.2 Changed Traceables in 17-10

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

▽

△

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

### B.1.3 Deleted Traceables in 17-10

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

**Table B.3: Deleted Traceables in 17-10**

### B.1.4 Added Constraints in 17-10

none

### B.1.5 Changed Constraints in 17-10

none

### B.1.6 Deleted Constraints in 17-10

none

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

### B.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 |
| [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 |

▽

△

| Number | Heading |
|---|---|
| [SWS_EM_02058] | State Transition Timeout |
| [SWS_EM_02102] | Memory control |
| [SWS_EM_02103] | CPU usage control |
| [SWS_EM_02104] | Core affinity |
| [SWS_EM_02106] | ResourceGroup assignment |
| [SWS_EM_02107] | Maximum heap |
| [SWS_EM_02108] | Maximum system memory usage |
| [SWS_EM_02109] | Process pre-mapping |
| [SWS_EM_02201] | ActivationReturnType Enumeration |
| [SWS_EM_02202] | ActivationTimeStampReturnType Enumeration |
| [SWS_EM_02210] | |
| [SWS_EM_02211] | DeterministicClient::DeterministicClient API |
| [SWS_EM_02215] | DeterministicClient::~DeterministicClient API |
| [SWS_EM_02216] | DeterministicClient::WaitForNextActivation API |
| [SWS_EM_02220] | DeterministicClient::RunWorkerPool API |
| [SWS_EM_02225] | DeterministicClient::GetRandom API |
| [SWS_EM_02230] | DeterministicClient::GetActivationTime API |
| [SWS_EM_02235] | DeterministicClient::GetNextActivationTime API |

**Table B.4: Added Traceables in 18-03**

### B.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 |

▽

△

| Number | Heading |
|---|---|
| [SWS_EM_01024] | Machine State Shutdown |
| [SWS_EM_01025] | Machine State Restart |
| [SWS_EM_01026] | State Change |
| [SWS_EM_01028] | Get State Information |
| [SWS_EM_01030] | Start of Process execution |
| [SWS_EM_01032] | Machine States Obtainment |
| [SWS_EM_01033] | Application start-up configuration |
| [SWS_EM_01034] | Deny State Change Request |
| [SWS_EM_01035] | Machine State Restart behavior |
| [SWS_EM_01036] | Machine State Shutdown behavior |
| [SWS_EM_01037] | Machine State Startup behavior |
| [SWS_EM_01041] | Scheduling FIFO |
| [SWS_EM_01042] | Scheduling Round-Robin |
| [SWS_EM_01043] | Scheduling Other |
| [SWS_EM_01050] | Start Dependent Processes |
| [SWS_EM_01051] | Shutdown Processes |
| [SWS_EM_01053] | Application State Running |
| [SWS_EM_01055] | Initiation of Process termination |
| [SWS_EM_01058] | Shutdown of the Operating System |
| [SWS_EM_01059] | Restart of the Operating System |
| [SWS_EM_01060] | Shutdown state change behavior |
| [SWS_EM_01061] | Override State Interrupt |
| [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 B.5: Changed Traceables in 18-03**

### B.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 |
| [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 B.6: Deleted Traceables in 18-03**

### B.2.4 Added Constraints in 18-03

none

### B.2.5 Changed Constraints in 18-03

none

### B.2.6 Deleted Constraints in 18-03

none

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

### B.3.1 Added Traceables in 18-10

none

### B.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 |

▽

△

| Number | Heading |
|---|---|
| [SWS_EM_01069] | Self-terminating Process State |
| [SWS_EM_01070] | Acknowledgement of termination request |
| [SWS_EM_01071] | Initiation of Process self-termination |
| [SWS_EM_01072] | Process Argument Zero |
| [SWS_EM_01074] | Short form arguments with option value |
| [SWS_EM_01075] | Short form Arguments without option value |
| [SWS_EM_01076] | Long form Arguments with option value |
| [SWS_EM_01077] | Long form Arguments without option value |
| [SWS_EM_01107] | Function Group configuration |
| [SWS_EM_01109] | Misconfigured Process instances |
| [SWS_EM_01110] | Off States |
| [SWS_EM_02000] | ExecutionState Enumeration |
| [SWS_EM_02001] | |
| [SWS_EM_02002] | ExecutionClient::~ExecutionClient API |
| [SWS_EM_02003] | ExecutionClient::ReportExecutionState API |
| [SWS_EM_02030] | ExecutionClient::ExecutionClient API |
| [SWS_EM_02044] | State Change in Progress |
| [SWS_EM_02049] | State Change Failed |
| [SWS_EM_02070] | ExecutionReturnType Enumeration |
| [SWS_EM_02109] | Process pre-mapping |
| [SWS_EM_02210] | |
| [SWS_EM_NA] | |

**Table B.7: Changed Traceables in 18-10**

### B.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 B.8: Deleted Traceables in 18-10**

### B.3.4 Added Constraints in 18-10

none

### B.3.5 Changed Constraints in 18-10

none

### B.3.6 Deleted Constraints in 18-10

none

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

### B.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 B.9: Added Traceables in R19-03**

### B.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 |

▽

△

| Number | Heading |
|---|---|
| [SWS_EM_02000] | |
| [SWS_EM_02001] | |
| [SWS_EM_02201] | |
| [SWS_EM_02202] | |
| [SWS_EM_02210] | |
| [SWS_EM_02241] | Machine State Startup Completion |
| [SWS_EM_02245] | Dependency resolution during state change |
| [SWS_EM_02246] | Process specific Environment Variables |

**Table B.10: Changed Traceables in R19-03**

### B.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 B.11: Deleted Traceables in R19-03**

### B.4.4 Added Constraints in R19-03

none

### B.4.5 Changed Constraints in R19-03

none

### B.4.6 Deleted Constraints in R19-03

none

## B.5 Constraint and Specification Item History of this document according to AUTOSAR Release R19-11

### B.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 B.12: Added Traceables in R19-11**

### B.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 |
| [SWS_EM_01003] | Starting Process State |

▽

△

| Number | Heading |
|---|---|
| [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 |
| [SWS_EM_01301] | Cyclic Execution |

▽

△

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

**Table B.13: Changed Traceables in R19-11**

### B.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 B.14: Deleted Traceables in R19-11**

### B.5.4 Added Constraints in R19-11

none

### B.5.5 Changed Constraints in R19-11

none

### B.5.6 Deleted Constraints in R19-11

none