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

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-03

Document Change History

Date	Release	Changed by	Description
2018-03-29	18-03	AUTOSAR Release Management	• Deterministic Execution • Resource Limitation • State Management • Fault Tolerance elaboration
2017-10-27	17-10	AUTOSAR Release Management	• State Management elaboration, introduction of Function Groups • Recovery actions for Platform Health Management • Resource limitation and deterministic execution
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

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 Adaptive Platform.

Chapter 8 documents the Execution Management Application Programming Interface (*API*). The inter-functional cluster API is described in Appendix C.

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 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 Adaptive Platform. However, the 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 ECUs built using that as well as Machines using the Adaptive Platform.

1.2 Interaction with AUTOSAR Runtime for Adaptive

The set of programming interfaces to the Adaptive Applications is called AUTOSAR Runtime for Adaptive (ARA). The interfaces that constitute ARA include those of Execution Management specified in Chapter 8. Note that APIs accessed by Adaptive Platform applications use the inter-functional cluster API is described in Appendix C which is not part of ARA.

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

The Adaptive AUTOSAR Services are provided via mechanisms provided by the `Communication Management` functional cluster [1] of the `Adaptive Platform Foundation`. In order to use the Adaptive AUTOSAR Services, the functional clusters in the `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 [2] AUTOSAR Glossary or [3] TPS Manifest Specification.

Term	Description
Process	A process is a loaded instance of an Executable to be executed on a Machine.
Execution Dependency	Dependencies between Executable instances can be configured to define a sequence for starting and terminating them.
Execution Management	The element of the Adaptive Platform responsible for the ordered startup and shutdown of the Adaptive Platform and the Applications.
State Management	The element of the Execution Management defining modes of operation for Adaptive Platform. It allows flexible definition of functions which are active on the platform at any given time. Architecture and functionality of State Management are still under dicussion. State Management will be covered by a new functional cluster in a later release.
Machine State	The element of the State Management which characterize the current status of the machine. It defines a set of active Applications for any certain situation. The set of Machine States is machine specific and it will be deployed in the Machine Manifest. Machine States are mainly used to control machine lifecycle (startup/shut-down/restart) and platform-level processes.
Function Group State	The element of State Management that characterizes the current status of a set of (functionally coherent) user-level Applications. The set of Function Groups and their Function Group States is machine specific and are deployed as part of the Machine Manifest.
Time Determinism	The results of a calculation are guaranteed to be available before a given deadline.
Data Determinism	The results of a calculation only depend on the input data and are reproducible, assuming a given initial internal state.
Full Determinism	Combination of Time and Data Determinism.

Table 2.1: Technical Terms

3 Related documentation

3.1 Input documents

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

[1] Specification of Communication Management
AUTOSAR_SWS_CommunicationManagement

[2] Glossary
AUTOSAR_TR_Glossary

[3] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[4] Requirements on Execution Management
AUTOSAR_RS_ExecutionManagement

[5] Requirements on Operating System Interface
AUTOSAR_RS_OperatingSystemInterface

[6] Requirements on Persistency
AUTOSAR_RS_Persistency

[7] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology

[8] 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

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

3.2 Related standards and norms

See chapter 3.1.

3.3 Related specification

See chapter 3.1.

4 Constraints and assumptions

4.1 Known limitations

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

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

Section 7.6 Deterministic Execution and Section 7.7 Resource Limitation – 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 7.8 Fault Tolerance – this section is incomplete and the topics of error handling within `Execution Management` will be expanded in a future release.

Section 7.4.5.1 State Management – This section will be removed as soon as a dedicated State Management specification document is available.

The following functionality is not specified in this release:

- ECU/VM reset interface ([RS_EM_00110]).
- `Application` integrity management ([RS_EM_00003].
- `Application` authentication and authorization ([RS_EM_00004].
- Container Support.

Appendix A details requirements from `Execution Management` Requirement Specification [4] that are not elaborated within this specification. The presence of these requirements in this document ensures that the requirement tracing is complete and also provides an indication of how `Execution Management` will evolve in future releases of the `Adaptive Platform`.

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

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other modules

5.1 Platform dependencies

5.1.1 Operating System Interface

Execution Management is dependent on the Operating System Interface [5]. The OSI is used to control specific aspects of Application execution, for example, to set scheduling parameters or to execute an Application.

5.1.2 Persistency

Execution Management is dependent on the Persistency [6] functional cluster. Persistency is used to access persistent storage and Manifest information.

5.2 Other dependencies

Currently, there are no other library dependencies.

6 Requirements tracing

The following tables reference the requirements specified in [4] and links to the fulfillment of these. Please note that if column "Satisfied by" is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_EM_00002]	Execution Management shall set-up one process for the execution of each Executable instance	[SWS_EM_01014] [SWS_EM_01015] [SWS_EM_01039] [SWS_EM_01040] [SWS_EM_01041] [SWS_EM_01042] [SWS_EM_01043]
[RS_EM_00003]	Execution Management shall support the checking of the integrity of Executables at startup of Executable.	[SWS_EM_NA]
[RS_EM_00004]	Execution Management shall support the authentication and authorization of Executables at startup of Executable	[SWS_EM_NA]
[RS_EM_00005]	Execution Management shall support the configuration of OS resource budgets for Executable and groups of Executables	[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 Executable threads to a specified set of processor cores.	[SWS_EM_02104]
[RS_EM_00009]	Only Execution Management shall start Executables	[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]
[RS_EM_00011]	Execution Management shall support self-initiated graceful shutdown of Executable instances	[SWS_EM_01005]
[RS_EM_00013]	Execution Management shall support configurable recovery actions	[SWS_EM_01016] [SWS_EM_01018] [SWS_EM_01061] [SWS_EM_01062] [SWS_EM_01063] [SWS_EM_01064]
[RS_EM_00050]	Execution Management shall perform system-wide coordination of Processes	[SWS_EM_NA]
[RS_EM_00051]	Execution Management shall provide functions to the Executable for configuring external trigger conditions for its activities	[SWS_EM_NA]
[RS_EM_00052]	Execution Management shall provide functions to the Executable for configuring cyclic triggering of its activities	[SWS_EM_01301] [SWS_EM_01302] [SWS_EM_02201] [SWS_EM_02210] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02216]

Requirement	Description	Satisfied by
[RS_EM_00053]	Execution Management shall provide functions to support deterministic redundant execution of Executables	[SWS_EM_01305] [SWS_EM_01308] [SWS_EM_01310] [SWS_EM_01311] [SWS_EM_01312] [SWS_EM_01313] [SWS_EM_02202] [SWS_EM_02210] [SWS_EM_02211] [SWS_EM_02215] [SWS_EM_02220] [SWS_EM_02225] [SWS_EM_02230] [SWS_EM_02235]
[RS_EM_00100]	Execution Management shall support the ordered startup and shutdown of Executables	[SWS_EM_01000] [SWS_EM_01001] [SWS_EM_01050] [SWS_EM_01051]
[RS_EM_00101]	Execution Management shall support State Management functionality	[SWS_EM_01013] [SWS_EM_01023] [SWS_EM_01024] [SWS_EM_01025] [SWS_EM_01026] [SWS_EM_01028] [SWS_EM_01032] [SWS_EM_01033] [SWS_EM_01034] [SWS_EM_01035] [SWS_EM_01036] [SWS_EM_01037] [SWS_EM_01044] [SWS_EM_01058] [SWS_EM_01059] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01068] [SWS_EM_01107] [SWS_EM_01108] [SWS_EM_01109] [SWS_EM_01110] [SWS_EM_01111] [SWS_EM_02044] [SWS_EM_02049] [SWS_EM_02050] [SWS_EM_02056] [SWS_EM_02057] [SWS_EM_02058] [SWS_EM_02070]
[RS_EM_00103]	Execution Management shall support Process lifecycle management	[SWS_EM_01002] [SWS_EM_01003] [SWS_EM_01004] [SWS_EM_01005] [SWS_EM_01006] [SWS_EM_01053] [SWS_EM_01055] [SWS_EM_01069] [SWS_EM_01070] [SWS_EM_01071] [SWS_EM_02000] [SWS_EM_02001] [SWS_EM_02002] [SWS_EM_02003] [SWS_EM_02030]
[RS_EM_00110]	Execution Management shall support diagnostic reset cause	[SWS_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 behaviour of Execution Management.

- Section 7.1 presents an introduction to key terms within Execution Management focusing on the relationship between Application, Executable, and Process.
- Section 7.2 covers the core Execution Management run-time responsibilities including the start of Applications.
- Section 7.3 describes the lifecycle of Applications including Process state transitions and startup / shutdown sequences.
- Section 7.4 covers several topics related to State Management within Execution Management including Machine, Application and Function Group state management.
- Section 7.5 describes how Application error recovery actions are specified during integration.
- Section 7.6 documents support provided by Execution Management *Deterministic* execution such that given the same input and internal state, a calculation will always produce the same output.
- Section 7.7 describes how Execution Management supports resource management including the limitation of usage of CPU and memory by an Application.
- Section 7.8 provides an introduction to Fault Tolerance strategies in general. This section will be expanded in a future release to describe how such strategies are realized within Execution Management.
- Section 7.9 covers development and deployment of Application specific information within the Manifest used by Execution Management to control execution of the Application.

7.1 Technical Overview

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

7.1.1 Terms

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

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

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

Process – Process (which technically is a POSIX process) is a started instance of an Executable.

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

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

7.1.2 Application

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

Applications can be located on user level above the middleware or can implement functional clusters of the Adaptive Platform (located on the level of the middleware), see [TPS_MANI_01009] in [3].

In general, an Application, whether user-level or platform-level, are treated the same by Execution Management and can use all mechanisms and APIs provided by the Operating System and other functional clusters of the Adaptive Platform.

However in doing so it potentially restricts its portability to other implementations of the Adaptive Platforms.

7.1.3 Adaptive Application

An Adaptive Application is a specific type of Application. The implementation of an Adaptive Application fully complies with the AUTOSAR specification, i.e. it is restricted to the use of APIs standardized by AUTOSAR and needs to follow specific coding guidelines to allow reallocation between different implementations of the Adaptive Platform.

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

Figure 7.1 shows the different types of Applications.

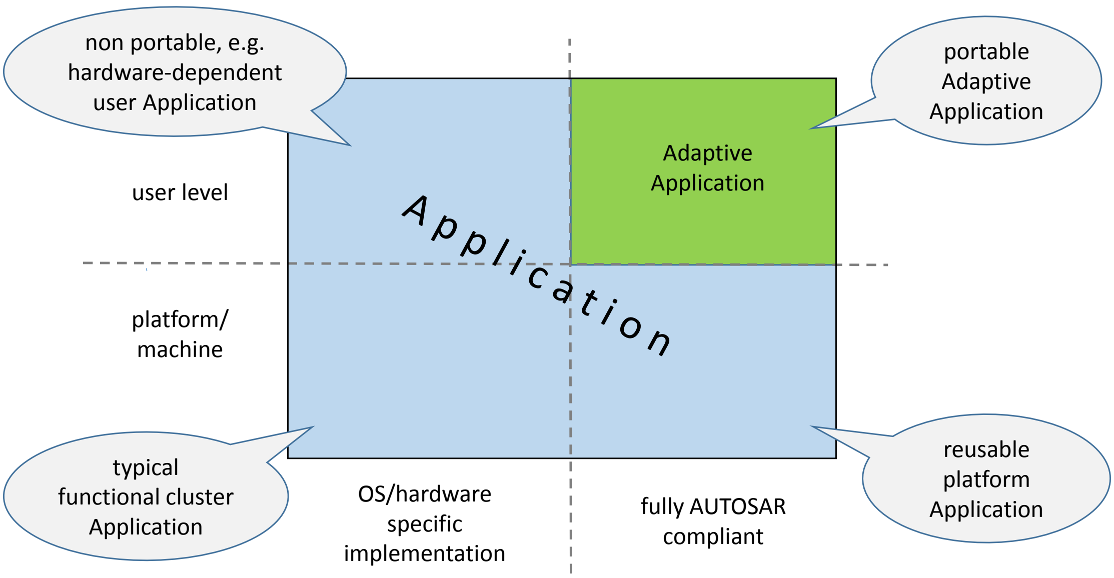

Figure 7.1: Types of Applications

An Adaptive Application is the result of functional development and is the unit of delivery for Machine specific configuration and integration. Some contracts (e.g. concerning used libraries) and Service Interfaces to interact with other Adaptive Applications need to be agreed on beforehand. For details see [7].

7.1.4 Executable

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

The lifecycle of `Executables` usually consists of:

Process Step	Software	Meta Information
Development and Integration	Linked, configured and calibrated binary for deployment onto the target `Machine`. The binary might contain code which was generated at integration time.	`Application Manifest`, see 7.1.6 and [3], and `Service Instance Manifest` (not used by Execution Management).
Deployment and Removal	Binary installed on the target `Machine`. Previous version (if any) removed.	Processed Manifests, stored in a platform-specific format which is efficiently readable at `Machine` startup.
Execution	`Process` started as instance of the binary.	The Execution Management uses contents of the Processed Manifests to start up and configure each `Process` individually.

Table 7.1: `Executable` Lifecycle

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

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

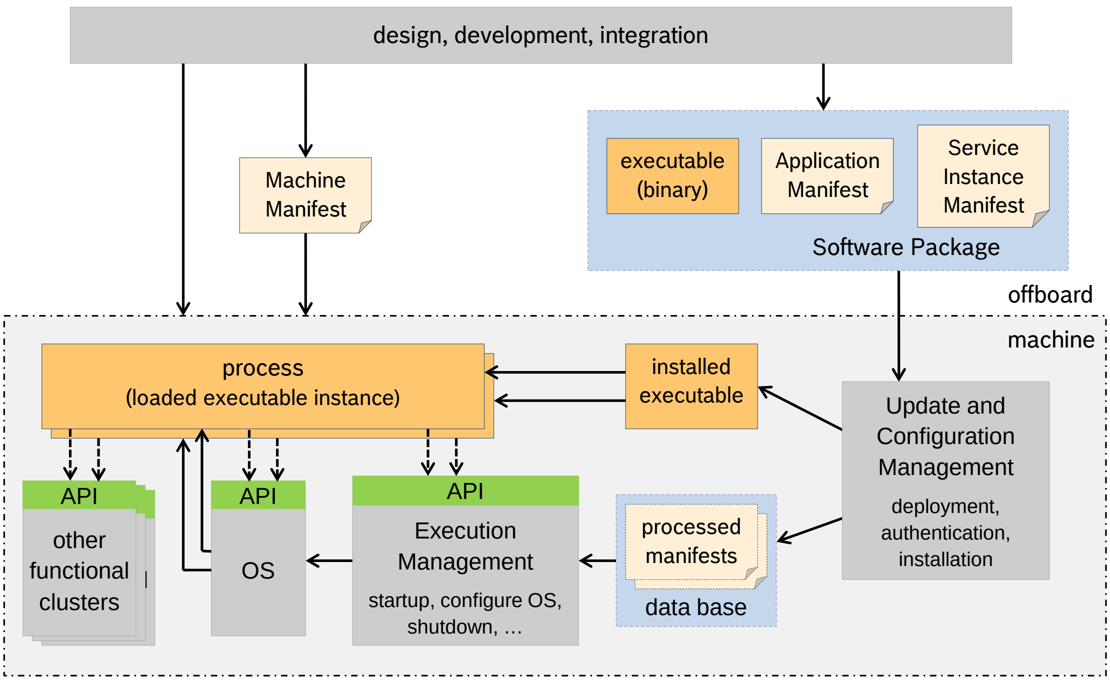

Figure 7.2: Executable Lifecycle from deployment to execution

7.1.5 Process

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

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

7.1.6 Application Manifest

An Application Manifest is created together with a Service Instance Manifest (not used by Execution Management) at integration time and deployed onto a Machine together with the Executable it is attached to. It supports the integration of the Executable code and describes in a standardized way the machine-specific configuration of Process properties (startup parameters, resource group assignment, priorities etc.).

The Application Manifest consists of parts of the Application design information which is provided by the application developer in an application description, and additional machine-specific information which is added at integration time. For details on the Application Manifest contents see Section 7.9. A formal specification can be found in [3].

Each instance of an Executable binary, i.e. each started Process, is individually configurable, with the option to use a different configuration set per Machine State or per Function Group State (see Section 7.4 and [TPS_MANI_01012], [TPS_MANI_01013], [TPS_MANI_01014], [TPS_MANI_01015], [TPS_MANI_01059], [TPS_MANI_01017] and [TPS_MANI_01041]).

7.1.7 Machine Manifest

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

The contents of a Machine Manifest includes the configuration of Machine properties and features (resources, safety, security, etc.), e.g. configured Machine States and Function Group States, resource groups, access right groups, scheduler configuration, SOME/IP configuration, memory segmentation. For details see [3].

7.1.8 Manifest format

The Application Manifests and the Machine Manifest can be transformed into a platform-specific format (called Processed Manifest), which is efficiently readable at Machine startup. The format transformation can be done either off board at integration time or at deployment time, or on the Machine (by Update and Configuration Management) at installation time.

7.2 Execution Management Responsibilities

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

Execution Management is started as part of the 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 Application Manifests.

[SWS_EM_01030] Start of Process execution ⌈ Execution Management shall be solely responsible for initiating execution of Processes. ⌋*(RS_EM_00009)*

Depending on the Machine State or on a Function Group State, deployed Executables are started during 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 contect of machine and/or function group state changes based on declared Execution Dependencies [SWS_EM_01050]. The dependencies are described in the Application 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. 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 Application Manifests.

7.3 Process Lifecycle Management

7.3.1 Process States

From the execution point of view, *Process States* characterize the lifecycle of any Process, i.e. of each instance of an Executable. Note that each Process is independent and therefore has its own *Process State*.

[SWS_EM_01002] Idle Process State ⌈ The **Idle** Process State shall be the Process State prior to creation of the 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 ⌈ The **Running** Process State shall apply to a Process after it has been scheduled and it has reported kRunning to Execution Management. ⌋*(RS_EM_00103)*

[SWS_EM_01005] Terminating Process State ⌈ The kTerminating Process State shall apply either after a Process has received the termination indication from Execution Management, or after it has decided to self-terminate and informed Execution Management. ⌋*(RS_EM_00103, RS_EM_00011)*

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

On entering the kTerminating Process State, the Process is expected to save persistent data, free all used Process internal resources and exit.

[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() command.

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 no longer run. This is relevant for one shot Processes which are supposed to run and terminate on their own as once they have reached their **Terminated** Process State they are to return to the **Idle** Process State without any external trigger.

[SWS_EM_01069] One-shot Process State ⌈ After a one-shot Process has terminated, Execution Management shall immediately set its Process State to Idle. ⌋*(RS_EM_00103)*

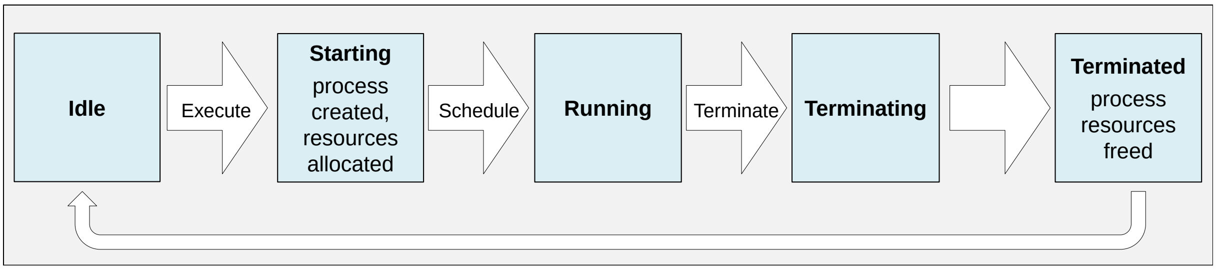

Figure 7.3: Process Lifecycle

7.3.2 Startup and Shutdown

7.3.2.1 Ordering

Execution Management can derive an ordering for the startup and shutdown of Processes within the State Management framework based on the declared Execution Dependencies. An Execution Dependency defines the provider of service(s) required by a Process before that Process can provide its own services. Hence Execution Management ensures the dependent Processes are in the state defined by the Execution Dependency before the Process with the dependency is started.

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

Example 7.1

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

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

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

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

Example 7.2

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

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

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

7.3.2.2 Arguments

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

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

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

[SWS_EM_01072] Application Argument Zero ⌈ Argument 0 shall be set to name of the Executable. ⌋*(RS_EM_00010)*

Execution Management supports simple arguments that take no value. All simple arguments begin with a single dash (-) which is not include in the StartupOption.optionName.

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

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

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

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

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

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

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

7.3.3 Startup Sequence

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

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

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

Please see Section 7.9.1.

Figure 7.4 shows the overall startup sequence.

[1] Typically the *init* process

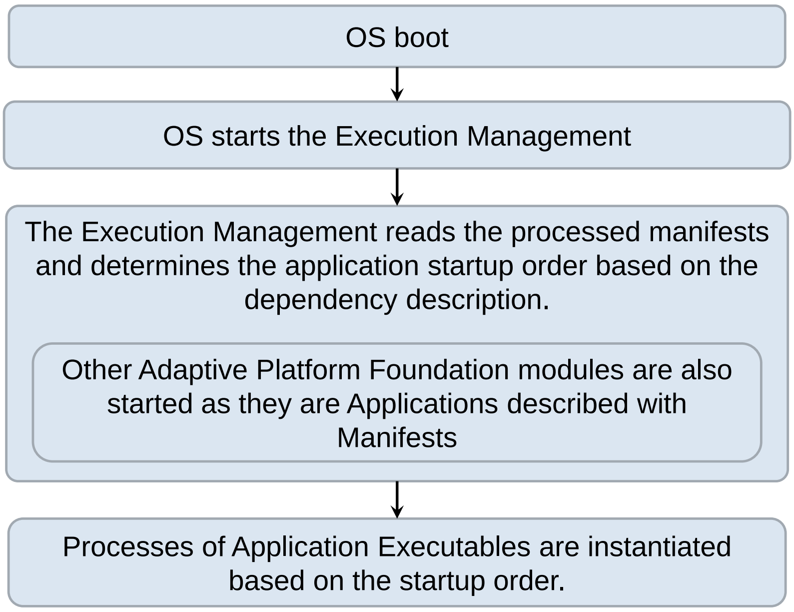

Figure 7.4: Startup sequence

7.3.3.1 Execution Dependency

Execution Management provides support to the Adaptive Platform for ordered startup and shutdown of Applications. This ensures that Applications are started before dependent Applications use the services that they provide and, likewise, that Applications are shutdown only when their provided services are no longer required. In this release, this only applies to platform-level Applications at machine startup and shutdown, see [constr_1484] in [3].

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

User-level Applications use service discovery mechanisms of the Communication Management and should not rely on Execution Dependencies. Which Processes are running depends on the current Machine State and on the current Function Group States, see 7.4. The integrator must 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 *Application State* (see 7.4.2) to be able to offer their services to other Processes, the dependency information shall also refer to *Application State* of the Process specified as dependency. With that in mind, the dependency information may be represented as a pair like: `<Process>.<ApplicationState>`. For more details regarding the *Application States* refer to Section 7.4.2.

The following dependency use-cases have been identified:

- In case Process B has a simple dependency on Process A, the *Running Application State* of Process A is specified in the dependency section of Process B's Application Manifest.
- In case Process B depends on One-Shot Process A, the *Terminated Application State* of Process A is specified in the dependency section of Process B's Application Manifest.

Version information within the Application Manifest is required since a consuming Executable and its required services might not be compatible with all versions of the producing Executable and its provided services. This also applies to the Processes which are instantiated from these Executables. An example for the definition of the version information attached to several Executables can be found in Listing 7.1.

Listing 7.1: Example for Executable versions

```
<AR-PACKAGE>
  <SHORT-NAME>Executables</SHORT-NAME>
  <ELEMENTS>
    <EXECUTABLE>
      <SHORT-NAME>RadarSensorVR</SHORT-NAME>
      <VERSION>1.0.3</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>RadarSensorVL</SHORT-NAME>
      <VERSION>1.0.4</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>Diag</SHORT-NAME>
      <VERSION>1.0.0</VERSION>
    </EXECUTABLE>
    <EXECUTABLE>
      <SHORT-NAME>SensorFusion</SHORT-NAME>
      <VERSION>1.0.2</VERSION>
    </EXECUTABLE>
  </ELEMENTS>
</AR-PACKAGE>
```

An example for the definition of the Process dependency information can be found in Listing 7.2

Listing 7.2: Example for Executable dependency

```
<PROCESS>
  <SHORT-NAME>SensorFusion</SHORT-NAME>
  <EXECUTABLE-REF DEST="EXECUTABLE">/Executables/SensorFusion</EXECUTABLE-
     REF>
  <MODE-DEPENDENT-STARTUP-CONFIGS>
    <MODE-DEPENDENT-STARTUP-CONFIG>
      <EXECUTION-DEPENDENCYS>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
```

```xml
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
               DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVR/
               ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
               PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
               DECLARATION-GROUP-PROTOTYPE">/Processes/RadarSensorVL/
               ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
               PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
        <EXECUTION-DEPENDENCY>
          <APPLICATION-MODE-IREF>
            <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST="MODE-
               DECLARATION-GROUP-PROTOTYPE">/Processes/Diag/
               ApplicationStateMachine</CONTEXT-MODE-DECLARATION-GROUP-
               PROTOTYPE-REF>
            <TARGET-MODE-DECLARATION-REF DEST="MODE-DECLARATION">/
               ModeDeclarationGroups/ApplicationStateMachine/Running</
               TARGET-MODE-DECLARATION-REF>
          </APPLICATION-MODE-IREF>
        </EXECUTION-DEPENDENCY>
      </EXECUTION-DEPENDENCYS>
      <STARTUP-CONFIG-REF DEST="STARTUP-CONFIG">/StartupConfigSets/
         StartupConfigSet_AA/SensorFusion_Startup</STARTUP-CONFIG-REF>
    </MODE-DEPENDENT-STARTUP-CONFIG>
  </MODE-DEPENDENT-STARTUP-CONFIGS>
</PROCESS>
```

Processes are only started by 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 or at machine startup/shutdown.

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

Example 7.3

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

7.4 State Management

7.4.1 Overview

State Management provides a mechanism to define the operational state of an Adaptive Platform. The Application Manifest allows to define in which states the Processes have to run (see [3]). As mentioned before, a Process is an instance of an Executable, which is part of an Application. State Management grants 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:

- Application State, see 7.4.2
- Process State

  Process States are managed by an Execution Management internal state machine. For details see Section 7.3.1.
- Machine State, see 7.4.3
- Function Group State, see 7.4.4

An example for the interaction between these states will be shown in section 7.4.5.2.

7.4.2 Application State

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

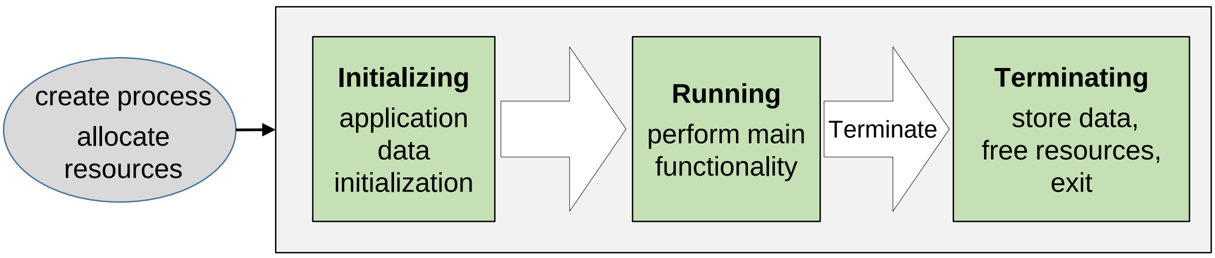

Figure 7.5: Application States

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

Please note that Service Discovery can introduce non-deterministic delays and thus is advised to be done after reporting kRunning state thus the Process may not have completed all its initialization when the kRunning state is reported.

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

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

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

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

During the Terminating state, the Process is expected to free internally used resources. The Process indicates completion of the Terminating state by simply exiting (with an appropriate exit code). Execution Management as the parent process can detect termination of the child process and take the appropriate platform-specific actions. For details on the response to “fault” error-codes, e.g. a non-zero exit code, will be defined in Section 7.8 in a future release of this document.

7.4.3 Machine State

Requesting and reaching a Machine State is, besides using Function Group States (see 7.4.4), one way to define the current set of running Processes. It is significantly influenced by vehicle-wide events and modes.

Each Application can declare in its Application Manifest in which Machine States it has to be running.

There are several mandatory Machine States specified in this document ([SWS_EM_01023], [SWS_EM_01024] and [SWS_EM_01025]) that have to be present on each machine. Additional Machine States can be defined on a machine specific basis and are therefore not standardized.

A ModeDeclaration for each required Machine State has to be defined in the Machine Manifest [TPS_MANI_03066].

[SWS_EM_01032] Machine States Obtainment ⌈ Execution Management shall obtain the Machine States from the Machine Manifest. ⌋*(RS_EM_00101)*

[SWS_EM_01044] Machine States Identification ⌈ The API specification shall use the shortName for identification of the Machine State. ⌋*(RS_EM_00101)*

The Machine States are determined and requested by the State Management functional cluster, see 7.4.5.1. For details on state change management see 7.4.6.

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

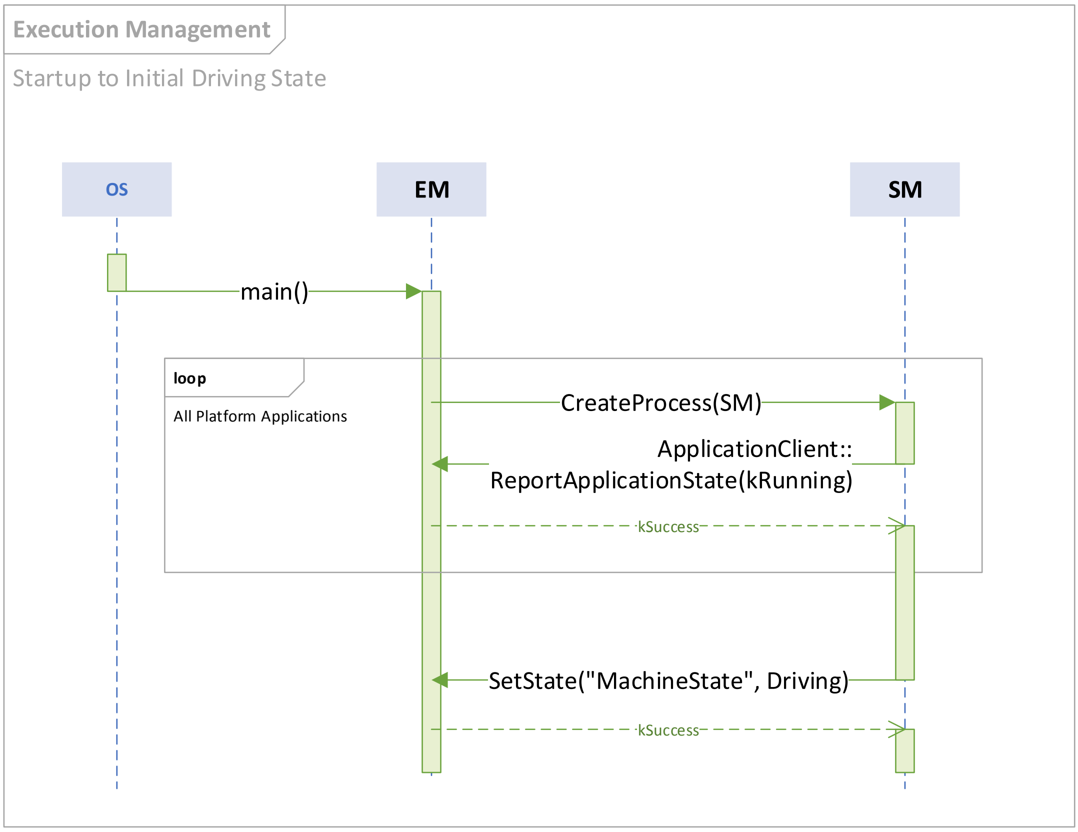

Figure 7.6: Start-up Sequence – from Startup to initial running state Driving

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

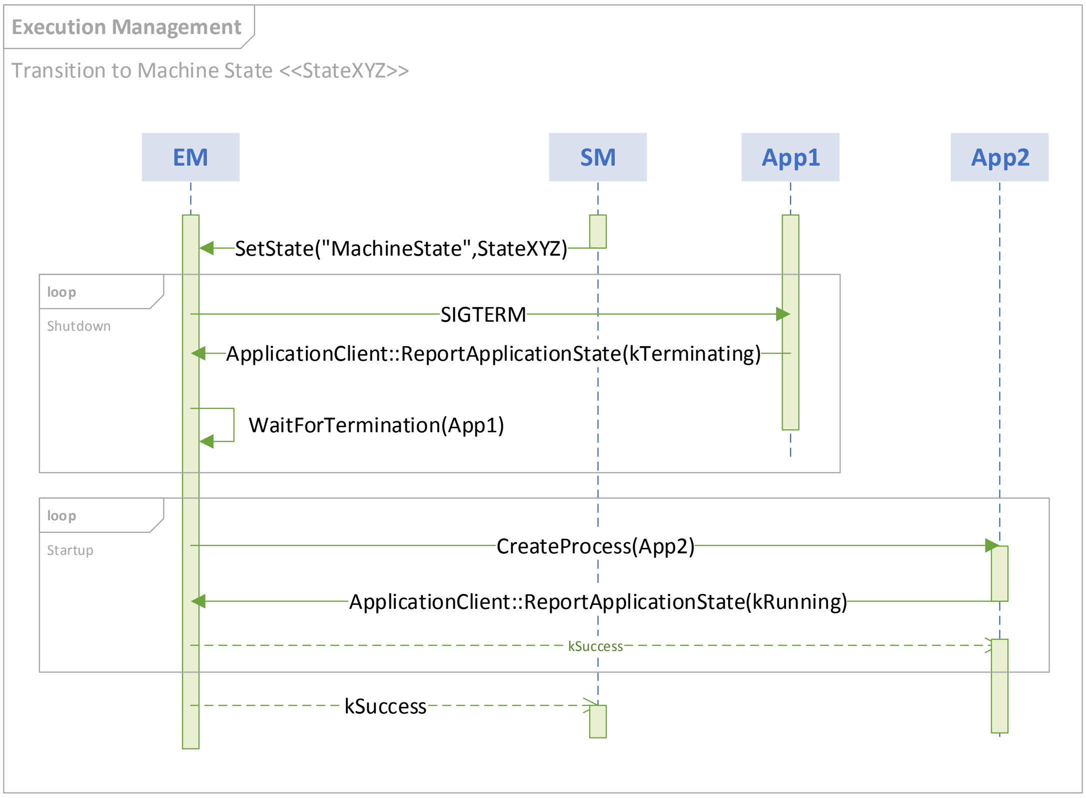

Figure 7.7: State Change Sequence – Transition to machine state `StateXYZ`

7.4.3.1 Startup

[SWS_EM_01023] Machine State Startup ⌈ The `Startup` `Machine State` shall be the first state to be active after the startup of `Execution Management`. ⌋ *(RS_EM_00101)*

[SWS_EM_01037] Machine State Startup behavior ⌈ The following behavior applies for the `Startup` `Machine State`:

- All `Processes` of platform-level `Applications` configured for `Startup` shall be started. `Processes` configured for `Startup` are based on the reference from the `Processes` to the `ModeDependentStartupConfig` in the role `Process.modeDependentStartupConfig` with the instanceRef to the `ModeDeclaration` in the role `ModeDependentStartupConfig.machineMode` that belongs to the `Startup` `Machine State`.
- For startup of `Processes`, the startup requirements of section 7.3 apply.
- `Execution Management` shall wait for all started `Processes` until their `Application State` `Running` is reported.

- If that is the case, Execution Management shall notify State Management that the Startup Machine State is ready to be changed.
- Execution Management shall not change the Machine State by itself until a new state is requested by State Management.

⌋(*RS_EM_00101*)

7.4.3.2 Shutdown

[SWS_EM_01024] Machine State Shutdown ⌈ The Shutdown Machine State shall be active after the Shutdown Machine State is requested by State Management. ⌋(*RS_EM_00101*)

[SWS_EM_01036] Machine State Shutdown behavior ⌈ The following behavior applies for the Shutdown Machine State:

- All Processes, including those of platform-level Applications, that have a Process State different than Idle or Terminated shall be shutdown.
- For shutdown of Processes, the shutdown requirements of section 7.3 apply.
- When Process State of all Processes is Idle or Terminated, all Processes configured for Shutdown shall be started. Processes configured for Shutdown are based on the reference from the Processes to the ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the Shutdown Machine State.

⌋(*RS_EM_00101*)

[SWS_EM_01058] Shutdown of the Operating System ⌈ There shall be at least one Application consisting of at least one Process that has a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the Shutdown Machine State. This Application shall contain the actual mechanism(s) to initiate shutdown of the Operating System. ⌋(*RS_EM_00101*)

7.4.3.3 Restart

[SWS_EM_01025] Machine State Restart ⌈ The Restart Machine State shall be active after the Restart Machine State is requested by State Management. ⌋(*RS_EM_00101*)

[SWS_EM_01035] Machine State Restart behavior ⌈ The following behavior applies for the Restart Machine State:

- All Processes, including those of platform-level Applications, that have a Process State different than Idle or Terminated shall be shutdown.
- For shutdown of Processes, the shutdown requirements of Section 7.3 apply.
- When Process State of all Processes is Idle or Terminated, all Processes configured for Restart shall be started. Processes configured for Restart are based on the reference from the Processes to the ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the Restart Machine State.

⌋*(RS_EM_00101)*

[SWS_EM_01059] Restart of the Operating System ⌈ There shall be at least one Application consisting of at least one Process that has a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with the instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode that belongs to the Restart Machine State. This Application shall contain the actual mechanism(s) to initiate restart of the Operating System. ⌋*(RS_EM_00101)*

7.4.4 Function Group State

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

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

In general, Machine States are used to control machine lifecycle (startup/shutdown/restart) and Processes of platform level Applications while Function Group States individually control Processes which belong to groups of functionally coherent user level Applications.

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

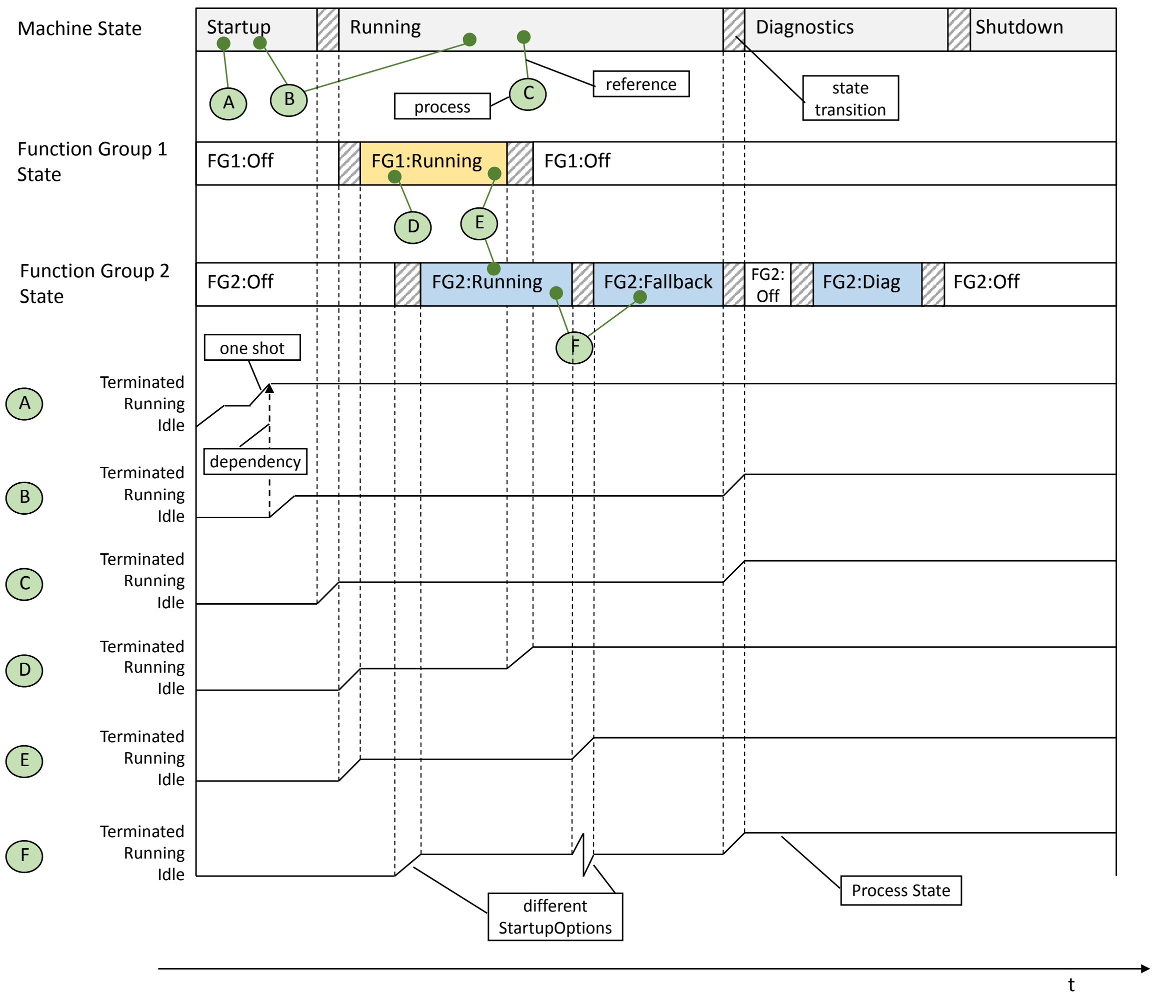

Figure 7.8: State dependent process control

- `Process` **A** references the `Machine State` `Startup`. It is a one shot `Process`, i.e. it terminates after executing once.
- `Process` **B** references `Machine States` `Startup` and `Running`. It depends on the termination of `Process` **A**, i.e. an `Execution Dependency` has been configured, as described in 7.3.3.1
- `Process` **C** references `Machine State` `Running` only. It terminates when `Machine State` `Diagnostics` is requested by `State Management`.
- `Process` **D** references `Function Group State` `FG1:Running` only.
- `Process` **E** references `FG1:Running` and `FG2:Running`. Because it references states of different `Function Groups`, it must use the same startup configuration (`StartupConfig`) in all states to avoid sequence dependent behaviour.

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

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

The Function Group States are determined and requested by the State Management functional cluster, see 7.4.5.1. For details on state change management see 7.4.6.

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

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

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

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

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

Processes reference in their Application Manifest the states in which they want to be executed. A state can be a Function Group State or a Machine State. For details see [3].

If a Process references Function Group States which belong to more than one Function Group, or if it references both Machine States and Function Group States, then only one startup configuration (StartupConfig) shall be configured, which is then valid for all referenced states.

This restriction prevents undefined behaviour, because if a Process references states of different Function Groups, which can be active simultaneously, the used startup configurations would depend on the sequence of the referenced Function Group States, if different startup configurations were used. Process **E** in Figure 7.8 is an example for such a Process.

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

The arbitrary state change sequence as shown in Figure 7.7 also applies to state changes of a Function Group - just replace "MachineState" by "FunctionGroup". 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.

7.4.5 State Management Architecture

7.4.5.1 State Management

Remark: The contents of this section is preliminary. This section will be removed as soon as a dedicated State Management specification document is available.

State Management is the functional cluster which is responsible for determining the current set of active Machine State and Function Group States, and for initiating State transitions by requesting them from Execution Management. Execution Management performs the State transitions and controls the actual set of running Processes, depending on the current States.

State Management is the central point where new Machine States and Function Group States can be requested and where the requests are arbitrated, including coordination of contradicting requests from different sources. Additional data and events might need to be considered for arbitration.

The State change requests can be issued by:

- Platform Health Management to trigger error recovery, e.g. to activate fallback functionality
- Diagnostics, to switch the system into diagnostic states
- Update and Configuration Management to switch the system into states where software or configuration can be updated
- Network Management to coordinate required functionality and network state
- authorized applications, e.g. a vehicle state manager which might be located in a different machine or on a different ECU

State Change requests can be issued by other Functional Clusters via Inter Functional Cluster (IFC) Interfaces, or ara::com service interfaces can be used to interact with State Management.

Since State Management functionality is critical, access from other Functional Clusters or Applications must be secured, e.g. by IAM (Identity and Access Management). State Management is monitored and supervised by Platform Health Management.

State Management provides interfaces to request information about current states.

State Management functionality is highly project specific, and AUTOSAR decided against specifying functionality like the Classic Platforms BswM for the Adaptive Platform. It is planned to only specify IFC interfaces and a set of basic service interfaces, and to encapsulate the actual arbitration logic into project specific code (e.g. a library), which can be plugged into the State Management framework and has standardized interfaces between framework and arbitration logic, so the code can be reused on different platforms.

The arbitration logic code might be individually developed or (partly) generated, based on standardized configuration parameters. These and other design decisions are still under discussion, and details will be provided at a later point in time.

An overview of the interaction of State Management, Execution Management and Applications is shown in Figure 7.9.

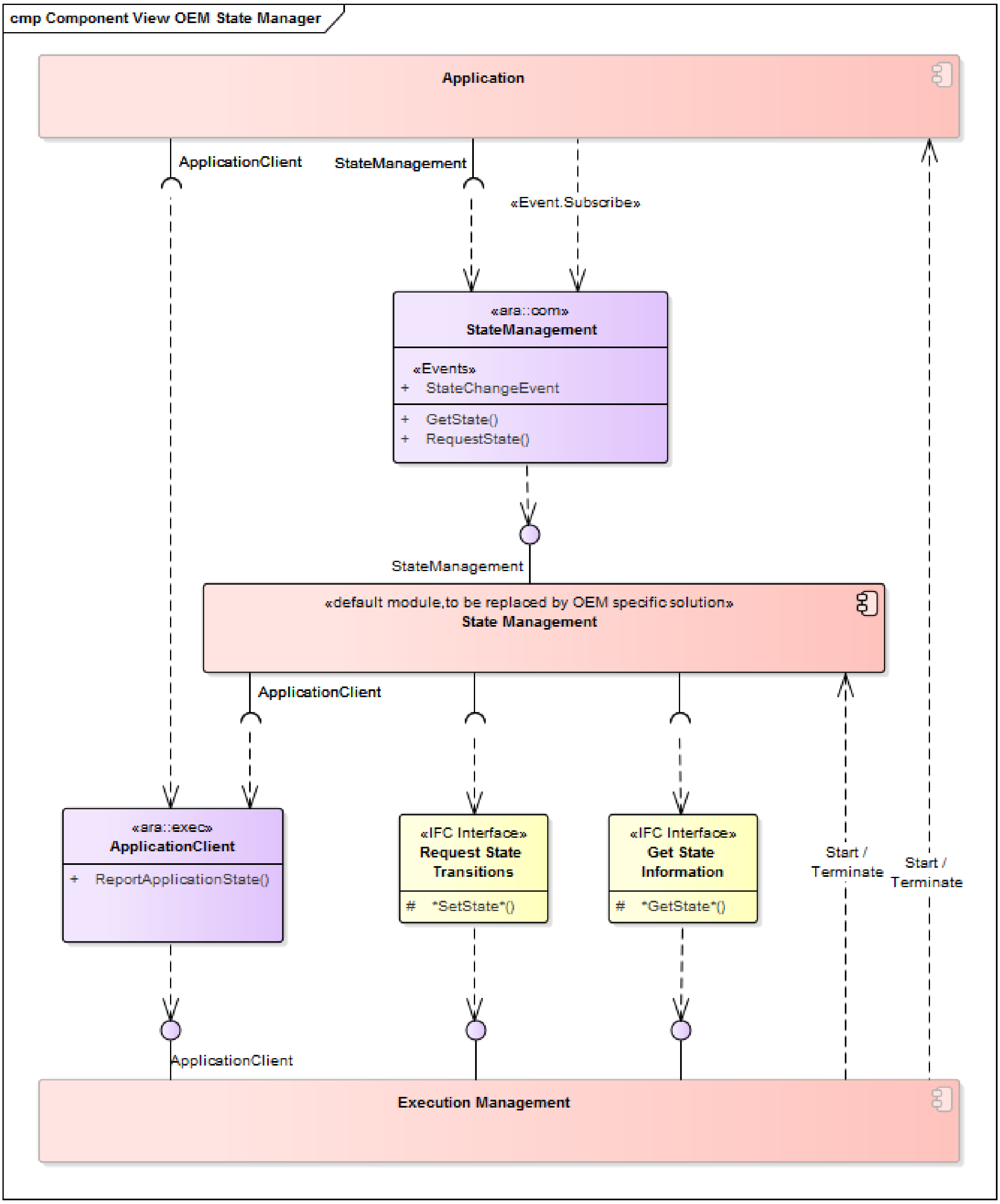

Figure 7.9: State Management Architecture

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

7.4.5.2 State Interaction

Figure 7.10 shows a simplified example for the interaction between different types of states. One can see the state transitions of a Function Group and the Process and Application States of one Process which references one state of this Function Group, ignoring possible delays and dependencies if several Processes were involved. The interaction is identical if the Process references a Machine State instead of a Function Group State.

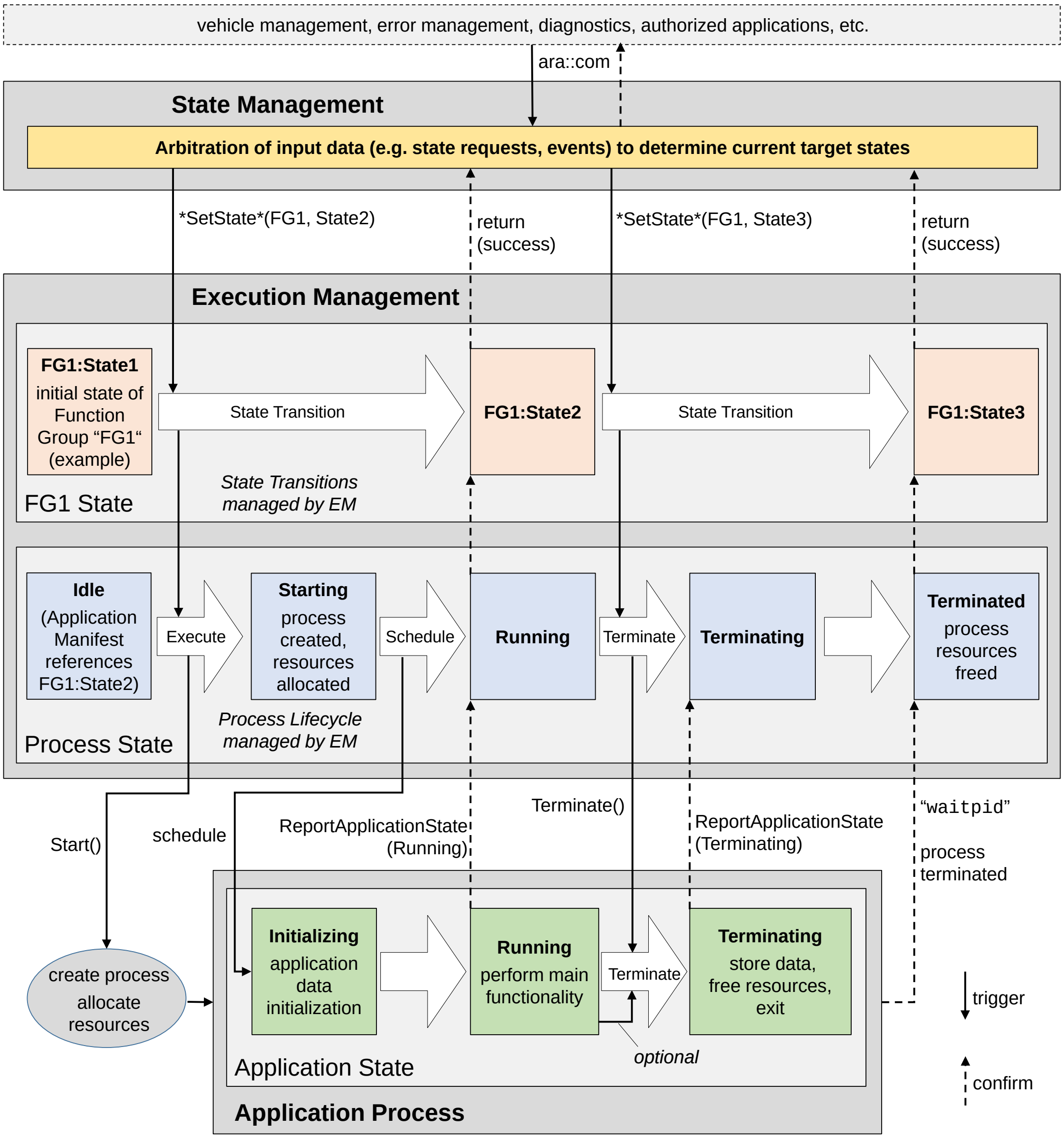

Figure 7.10: Interaction between states

7.4.6 State Change

State Management can request to change one or several Function Group States and/or the Machine State from Execution Management by passing pairs of <Function Group><requested State> as parameters, with Machine State being treated like any Function Group State.

[SWS_EM_01026] State Change ⌈ A state change request by State Management shall lead to immediate state transitions and hereof a state change to the requested Machine State and/or Function Group States. ⌋*(RS_EM_00101)*

State Management can request multiple Machine State and Function Group State changes sequentially by issuing several individual state change requests, or atomically within the same state change request, which leads to multiple coherent state changes. However, the following restriction applies to avoid undefined behaviour while the state transitions are performed by Execution Management:

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

[SWS_EM_02058] State Transition Timeout ⌈ If a timeout is detected when stopping or starting Processes at a state transition, Execution Management shall return an error code to the requester of the state changes ⌋*(RS_EM_00101)*

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

[SWS_EM_02056] State Change Failed ⌈ Execution Management shall return an error code to the requester of the state changes when other or unspecified errors occur at a state transition. ⌋*(RS_EM_00101)*

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

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

In the following requirements, the term

"the *Process* references a *State*"

means that a Process has in its Application Manifest an aggregation from the Process containing a ModeDependentStartupConfig in the role Process.modeDependentStartupConfig with an instanceRef to a ModeDeclaration in the role ModeDependentStartupConfig.machineMode or in the role ModeDependentStartupConfig.functionGroupMode that belongs to that *State*.

A *State* can be a Machine State or a Function Group State dependent on the used reference to a ModeDeclaration.

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

RequestedStates is the collection of the Function Group States of all configured Function Groups and the Machine State at the point in time when all ongoing state transitions are finished. (Remember that new state change requests are rejected until this point in time, see [SWS_EM_01034])

A *SingleReferenceProcess* references in its Application Manifest either Machine States or states of one Function Group only. In Figure 7.8 this would apply to all Processes except Process **E**.

A *MultiReferenceProcess* references in its Application Manifest more than one type of states, e.g. Machine States and Function Group States, or states of more than one Function Group. In Figure 7.8 this would apply to Process **E**. As explained in section 7.4.4, different startup configurations are not permitted in this case.

On a state change Execution Management is required to shutdown no longer active Processes ([SWS_EM_01060]). For shutdown the requirements of Section 7.3 apply.

[SWS_EM_01060] Shutdown state change behavior ⌈

For each *SingleReferenceProcess*, that

- references exactly one of the *CurrentStates*, and
- references none of the *RequestedStates*, and
- has a Process State different than [Idle or Terminated]

or

- references exactly one of the *CurrentStates*, and
- references exactly one of the *RequestedStates*, and
- has different aggregated StartupOptions in the role StartupConfig.startupOption, referenced by the ModeDependentStartupConfigs in the role ModeDependentStartupConfig.startupConfig
  - with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode or ModeDependentStartupConfig.functionGroupMode that belongs to the referenced *CurrentState*, and
  - with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode or ModeDependentStartupConfig.functionGroupMode that belongs to the referenced *RequestedState*.

and, for each *MultiReferenceProcess*, that

- references at least one of the *CurrentStates*, and
- references none of the *RequestedStates*, and
- has a Process State different than [Idle or Terminated]

the Process shall be shutdown. ⌋*(RS_EM_00101)*

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

Execution Management waits until the Process State of all affected Processes is Idle or Terminated.

[SWS_EM_01065] Shutdown state timeout monitoring behavior ⌈

Execution Management shall monitor the time required by the Processes to terminate – that is the Process State of the Process is Idle or Terminated. In case of a timeout ([TPS_MANI_03151]) the following set of actions shall be performed by Execution Management:

- Platform Health Management is notified about the timeout to initiate appropriate recovery actions.
- The timeout condition is reported back to the requester of the State transition to notify that the State change request cannot be fulfilled, see [SWS_EM_02058].

⌋*(RS_EM_00101)*

On a state change Execution Management is required to start Processes active in the new state. For startup the requirements of section 7.3 apply ([SWS_EM_01066]).

[SWS_EM_01066] Start state change behavior ⌈

For each *SingleReferenceProcess*, that

- references none of the *CurrentStates*, and
- references exactly one of the *RequestedStates*, and
- has a Process State that is [Idle or Terminated]

or

- references exactly one of the *CurrentStates*, and
- references exactly one of the *RequestedStates*, and
- has different aggregated StartupOptions in the role StartupConfig.startupOption, referenced by the ModeDependentStartupConfigs in the role ModeDependentStartupConfig.startupConfig

- with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode or ModeDependentStartupConfig.functionGroupMode that belongs to the referenced *CurrentState*, and
- and with an instanceRef to the ModeDeclaration in the role ModeDependentStartupConfig.machineMode or ModeDependentStartupConfig.functionGroupMode that belongs to the referenced *RequestedState*.

and, for each *MultiReferenceProcess*, that

- references none of the *CurrentStates*, and
- references at least one of the *RequestedStates*, and
- has a Process State that is [Idle or Terminated]

the Process shall be started. ⌋*(RS_EM_00101)*

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

Execution Management waits until the Process State of all affected Processes is Running.

Execution Management shall monitor the time required by the Processes to reach the Running state. For definition of the Process start-up timeout parameters in the Application Manifest see [TPS_MANI_03149].

[SWS_EM_01067] Confirm State Changes ⌈ In case the Processes report the Running state within the defined timeout interval ([TPS_MANI_03146]), Execution Management shall send a confirmation of the state change to the initiator of the state change. ⌋*(RS_EM_00101)*

[SWS_EM_01068] Report start-up timeout ⌈ In case of a timeout the following set of actions shall be performed by Execution Management:

- Platform Health Management is notified about the timeout to initiate appropriate recovery actions.
- The timeout condition is reported back to the requester of the State transition to notify that the State change request cannot be fulfilled, see [SWS_EM_02058].

⌋*(RS_EM_00101)*

7.4.7 State Information

[SWS_EM_01028] Get State Information ⌈ Execution Management shall provide an interface to retrieve the current Machine State or a Function Group State by passing a Function Group identifier as parameter, with “MachineState” being treated like any Function Group. ⌋(RS_EM_00101)

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

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

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

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

A table that summarized the requirements of this chapter can be found in Appendix C.2.3.

7.5 Application Recovery Actions

7.5.1 Overview

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

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

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

7.5.2 Recovery Actions

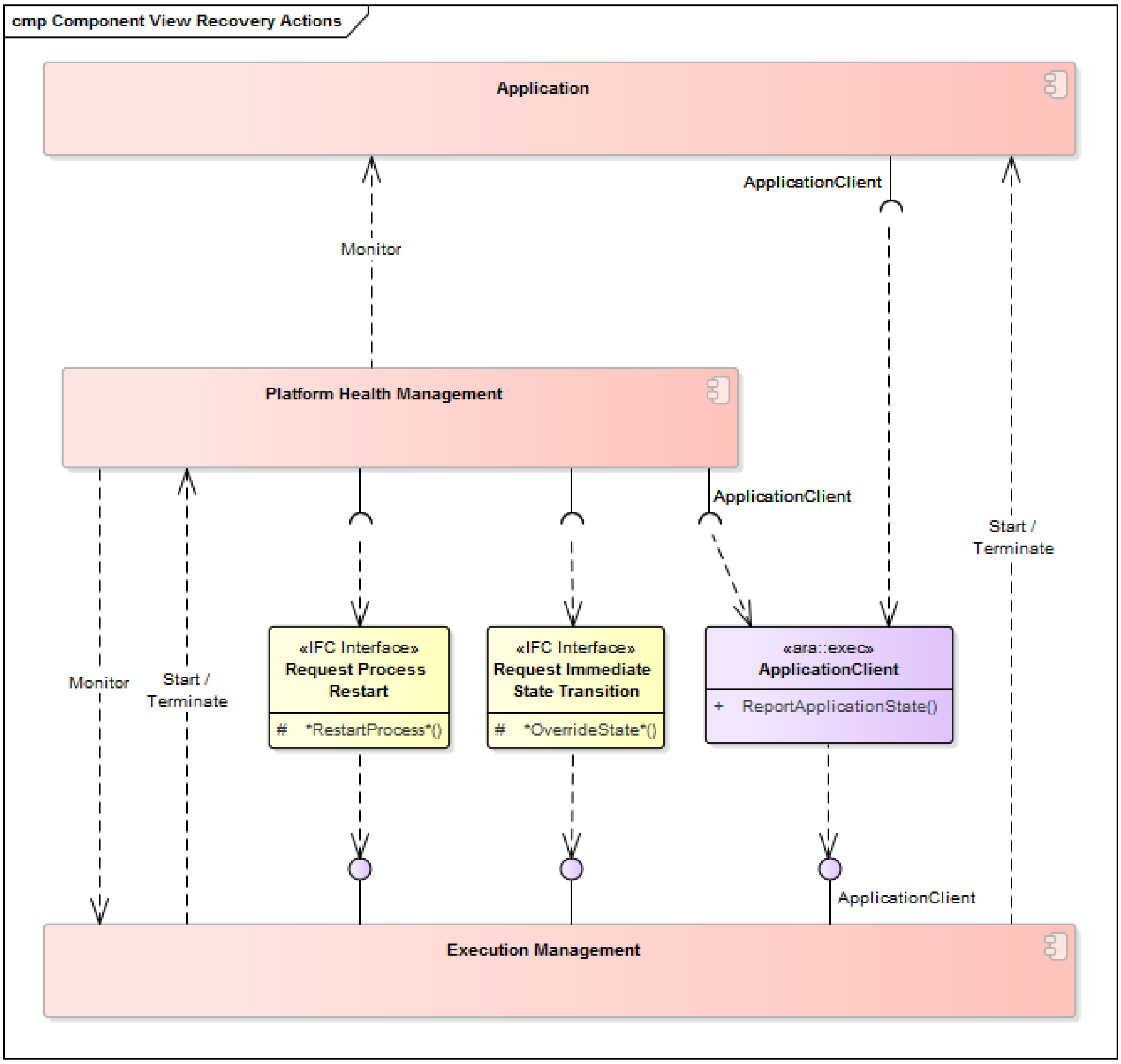

Figure 7.11: Adaptive Platform - Recovery Action Architecture

7.5.2.1 Restart Process

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

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

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

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

7.5.2.2 Override State

[SWS_EM_01018] Override State ⌈ Execution Management shall provide an inter functional cluster interface to force Execution Management to switch to specific Function Group States and/or to a specific Machine State on the request from the Platform Health Management. ⌋*(RS_EM_00013)*

[SWS_EM_01061] Override State Interrupt ⌈ An Override State request shall stop any currently "ongoing" state transition and process the "override" state changes. ⌋ *(RS_EM_00013)*

Please note that [SWS_EM_02056], [SWS_EM_02057] and [SWS_EM_02058] also apply for Override State requests.

Machine State and Function Group State changes can be requested individually or in parallel by the Platform Health Management.

The rules for state transitions as described in [SWS_EM_01060], [SWS_EM_01065], [SWS_EM_01066], [SWS_EM_01067], and [SWS_EM_01068] also apply for Override State requests. Please note that a termination request that may be required to be send to a Process, should be delayed until this Process reports its Running Application State.

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 worthwile 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 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.) must 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. However, if Time Determinism is requested, the resources must be guaranteed at any time, i.e. minimum and maximum resources are identical.

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

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

7.6.1.2 Data Determinism

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

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

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

7.6.1.3 Full Determinism

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

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

Non-deterministic behavior may arise from different reasons; for example insufficient computing resources, 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, sporadic unrelated events, race conditions, etc.

7.6.2 Redundant Deterministic Execution

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

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

Figure 7.12 shows a simplified example for a possible software lockstep architecture.

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

Execution Management provides DeterministicClient APIs to support control of the process-internal cycle, a deterministic worker pool, activation time stamps and random numbers. In case of software lockstep, the DeterministicClient interacts with an optional software lockstep framework to ensure identical behavior of the redundantly executed Processes. DeterministicClient interacts with Communication Management to synchronize data handling with cycle activation.

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

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

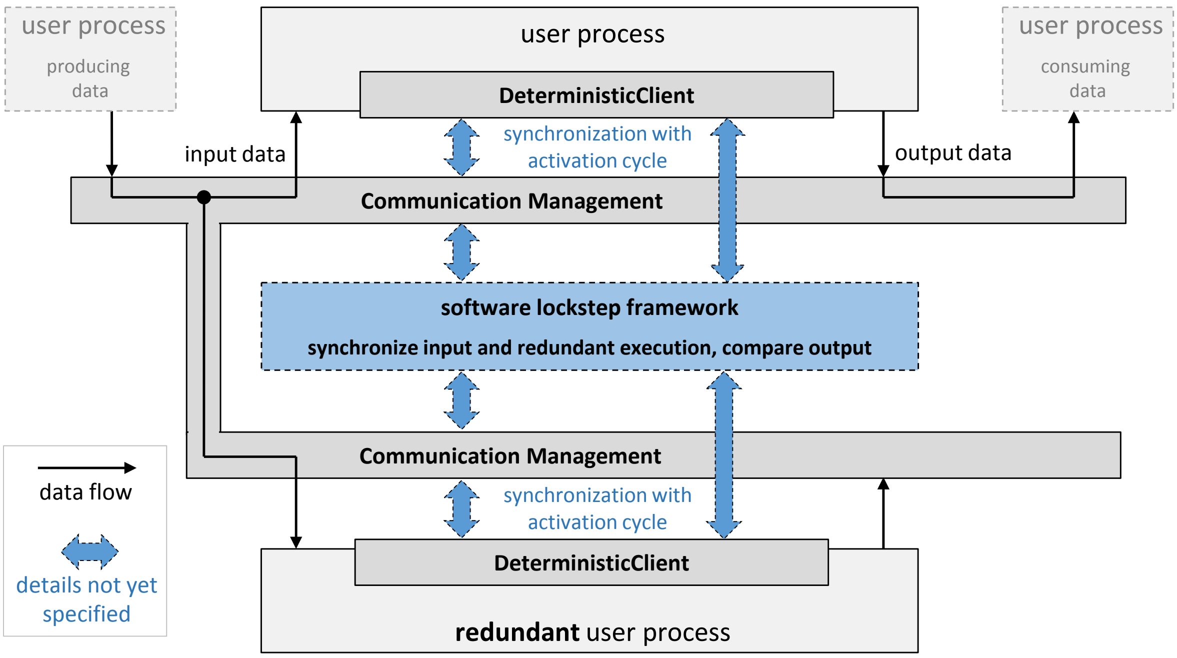

Figure 7.12: Software Lockstep in a typical data flow processing

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

Figure 7.13 zooms into one of the redundantly executed Processes.

The 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.

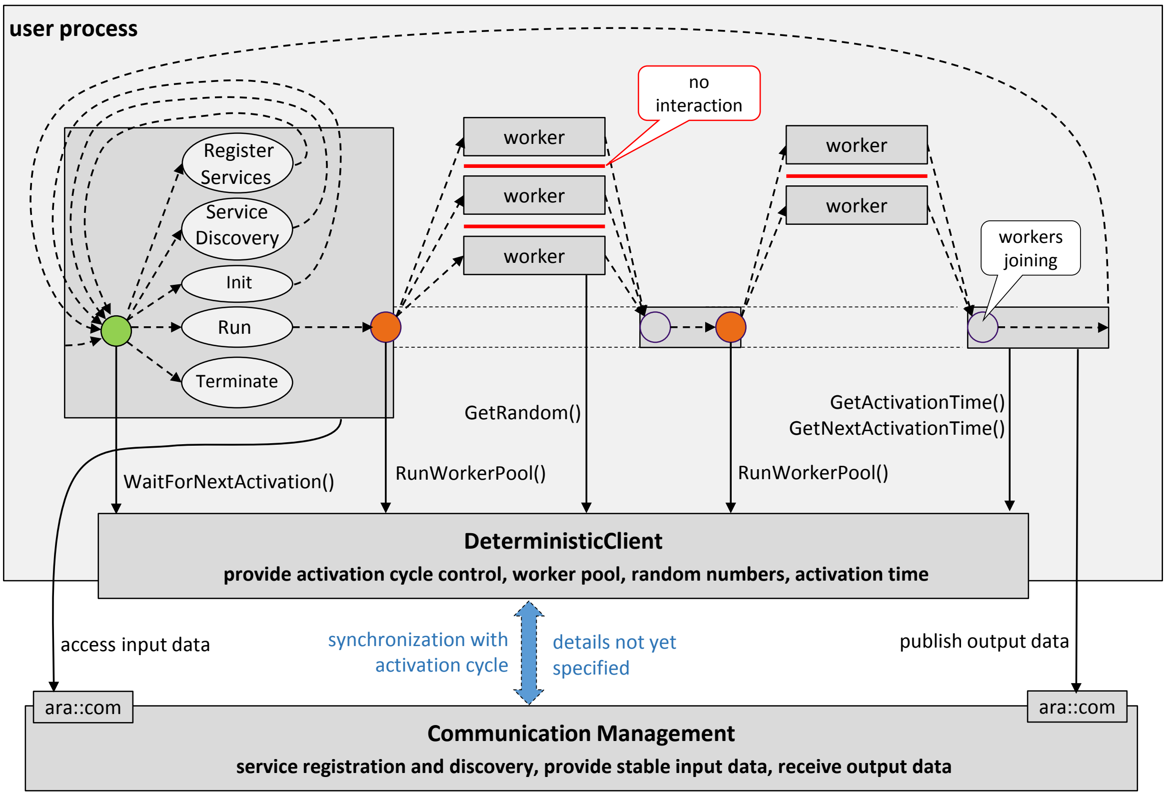

Figure 7.13: Cyclic Deterministic Execution

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 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 parallel execution of application functions (workers), which are not allowed to exchange any information while they are running, i.e. they don't access data which can be altered by other workers to avoid race conditions. The workers can physically run in parallel or sequentially in any order. For details see 7.6.3.2.

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

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

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

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

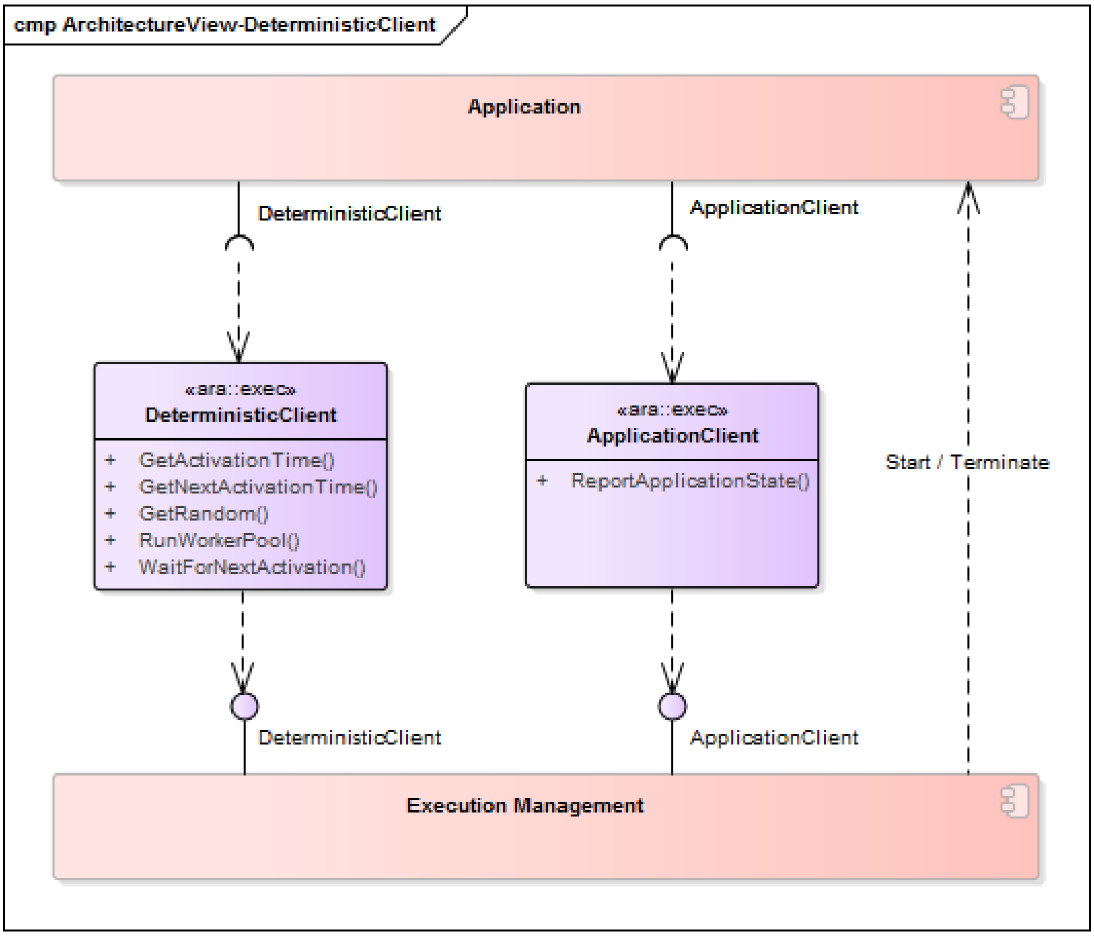

Figure 7.14: Deterministic Execution Interface

7.6.3 Cyclic Deterministic Execution

This section describes the APIs shown in Figure 7.13, 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. A return value controls the internal lifecycle (e.g. init, run, terminate) of the Process, see Figure 7.13.

[SWS_EM_01301] Cyclic Execution ⌈ Execution Management shall provide a blocking wait point API DeterministicClient::WaitForNextActivation. The Process executes one cycle when the wait point API returns and then calls the API again to wait for the next activation. ⌋*(RS_EM_00052)*

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

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

[SWS_EM_01302] Cyclic Execution Control ⌈ DeterministicClient::WaitForNextActivation shall return a code to control the execution mode of the calling Process. Possible modes are “Register Services”, “Service Discovery”, “Init”, “Run”, and “Terminate”. ⌋*(RS_EM_00052)*

The return codes are used to synchronize the behavior of the Processes in case of redundant execution. The Processes return to DeterministicClient::WaitForNextActivation after each of the usual sequential steps

- Register Services: The Process registers communication services (this must be the only occasion for performing service registering).
- Service Discovery: The Process does communication service discovery (this must be the only occasion for performing service discovery).
- Init: The Process initializes its internal data structures (once).
- Run: The Process performs one cycle of its normal cyclic execution.
- Terminate: The Process terminates.

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) must be explicitly visible for Execution Management.

Execution Management together with Communication Management initiate 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 must be available when a cycle starts and guaranteed to be deterministically consistent.

Configuration details (e.g. activation period) will be provided in a later release.

7.6.3.2 Worker Pool

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

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

The implementation and size of the worker pool is hidden from the user. The Integrator decides about the size (a configuration parameter "NumberOfWorkers" will be added in the next release of the Adaptive Platform specification) and the implementation.

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

To achieve Data Determinism, the parallel workers within a Process need to satisfy certain implementation properties, e.g. no exchange of data is allowed between the workers. For details see section 7.6.3.6. Other, more complex solutions which allow interaction between the workers would be possible, but they increase complexity, reduce utilization and transparency, and are error-prone regarding the deterministic behavior.

The worker pool runs within the Process context of the caller of this API. It is designed as part of Execution Management to guarantee the deterministic behavior by incorporating it in the DeterministicClient::WaitForNextActivation-cycle, where also the seeds for the pseudo random generation are provided (see 7.6.3.3).

DeterministicClient::RunWorkerPool registers a "worker" runnable object, along with its parameter object. The parameter contains a set of objects, which are processed in parallel by the same runnable object invoked from multiple workers in the pool. 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. (e.g. based on POSIX threads.) The deterministic distribution of the elements to individual workers is done by using the container iterator.

An example for the implementation of a “worker” runnable object can be found in section 7.6.3.7

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-runable-object (here the worker-runable-object would be e. g. the Kalman-filter).

The worker pool cannot be used to process multiple different tasks in parallel. The use of multiple potentially different explicit functions (workers) 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 `DeterministicClient::GetRandom` which provides “Deterministic” random numbers. ‘Deterministic” means, that the provided random numbers are identical for `Processes` which are executed redundantly, including within workers being processed by a worker pool (see [SWS_EM_01305]). ⌋*(RS_EM_00053)*

The random numbers are assigned to specific workers to allow deterministic redundant execution.

For the cyclic behavior of the workers, `Execution Management` uses a deterministic and unique pseudo random number concept.

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. 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 must 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 `DeterministicClient::GetActivationTime` which provides a deterministic timestamp that represents the point in time when the current cycle was activated by `DeterministicClient::WaitForNextActivation` (see [SWS_EM_01301]). Deterministic means, that the timestamps are identical for `Processes` which are executed redundantly. Subsequent calls within a cycle shall always return the same value. ⌋*(RS_EM_00053)*

[SWS_EM_01311] Activation Time Unknown ⌈ In case no previous call of DeterministicClient::WaitForNextActivation with return value kRun has occured when calling DeterministicClient::GetActivationTime, Execution Management shall return kNotAvailable. ⌋*(RS_EM_00053)*

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

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

7.6.3.5 Real-Time Resources

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

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

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

To describe budget needs, we use a normalized value *#Instructions* to specifiy runtime consumption on the target system.

#Instructions = runtime in sec * clock frequency / 1sec

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

The following parameters are relevant for describing the computing time budget needs of a `Process` which uses `DeterministicClient::RunWorkerPool`. They will be formally specified in the next release of the `Adaptive Platform` specification.

- *NumberOfInstructions* [#Instructions]

  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.

- *NumberOfWorkers*

  The most workers which can be used in parallel to speed up calculation, assuming enough physical worker cores were available on the machine.

- *Speedup* = sequental runtime / parallelized runtime

  Defines how much faster the calculations within one cycle can be finished if *NumberOfWorkers* are physically available.

- *SequentialInstructionsBegin* [#Instructions]

  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* [#Instructions]

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

Examples

Example 7.4

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

- *NumberOfInstructions* = 500
- *NumberOfWorkers* = 2
- *Speedup* = 1.3

- *SequentialInstructionsBegin* = 100
- *SequentialInstructionsEnd* = 125

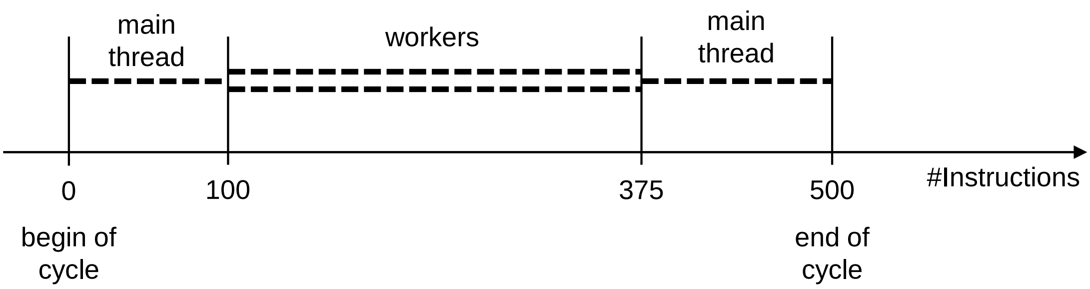

Figure 7.15: Worker pool used in middle of cycle

Example 7.5

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

- *NumberOfInstructions* = 200
- *NumberOfWorkers* = 2
- *Speedup* = 1
- *SequentialInstructionsBegin* = 200
- *SequentialInstructionsEnd* = 0

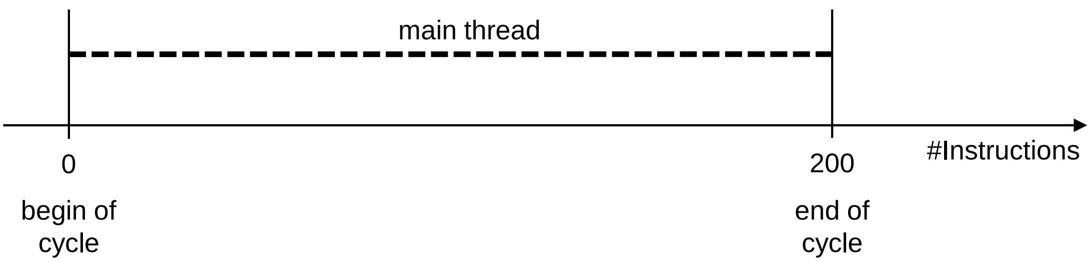

Figure 7.16: No benefit from worker pool

Example 7.6

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

- *NumberOfInstructions* = 200
- *NumberOfWorkers* = 3

- *Speedup* = 2.9
- *SequentialInstructionsBegin* = 0
- *SequentialInstructionsEnd* = 0

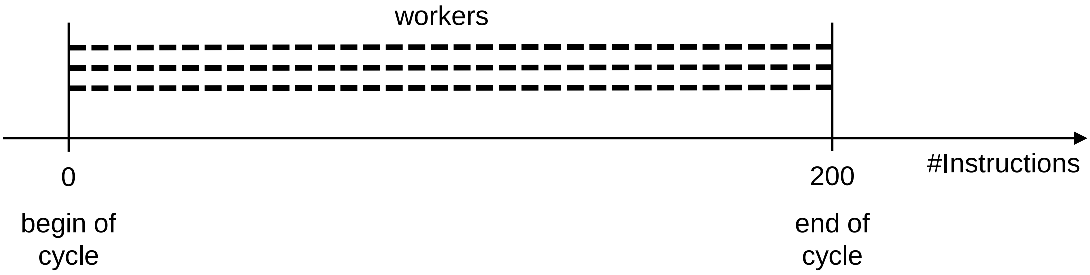

Figure 7.17: Full utilization of worker pool

7.6.3.6 Guidelines for implementation of deterministic user process

If the worker pool (see 7.6.3.2) is used, the container elements, i.e. the jobs to be computed, need to satisfy certain implementation rules to ensure Data Determinism.

- No exchange of data between workers, i.e. no communication. Individual workers must not access data that is influenced by other workers to avoid race conditions.

  Rationale: Timing between individual workers is not guaranteed. The Operating System is scheduling threads individually. Concurrent influencing of the same data will result in indeterminate results.

- No locks and synchronization points except common joins for all workers. (e.g. no Semaphores/Mutexes, no locking/blocking).

  Rationale: locking/blocking makes `Process` runtime in-deterministic. Workers are used to increase utilisation of runtime. If synchronization is needed, an explicit join of all workers is necessary.

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.

To ensure deterministic behavior, only a “deterministic subset” of all available POSIX PSE51 APIs and ara::com mechanisms are allowed to be used in a deterministic user `Process`. A detailed list of such APIs and mechanisms will be provided at a later point in time.

7.6.3.7 Implementation of Worker Pool users

Example of a worker-runnable:

```
class MyWorker1
:   public DeterministicClient::WorkerrunableBase<myContainer::
    value_type, MyWorker1>
{
public:
    void worker_runable(myContainer::value_type& container_element,
        DeterministicClient::WorkerThread& t)
    {
        // Get a unique and deterministic pseudo-random number}
        uint64_t random_number = t.GetRandom();
    }
};
```

Worker-thread object:

```
class DeterministicClient::WorkerThread
{
  // returns a deterministic pseudo-random number}
  // which is unique for each worker}
  uint64_t  GetRandom();

  ...
};
```

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

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

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

7.7.1 Resource Configuration

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

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

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

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

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

The following stakeholders are involved in resource management:

- Application Developer

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

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

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

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

- Integrator

  The integrator knows the specific platform and its available resources and constraints, as well as other applications which may run at the same time as the Processes to be configured. The integrator must assign available resources to the applications which can be active at the same time, which is closely related to State Management configuration, see section 7.4. If not enough resources are available at any given time to fulfill the maximum resource needs of all running Processes, assuming they are actually used by the Processes, several steps have to be considered:

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

Based on the above, all the resource configuration elements are to be configured during platform integration, most probably by the Integrator. To group these configuration elements, we define a ResourceGroup. It may have several properties configured to enable restricting Applications running in the group. Subsequently, each Process must belong to a ResourceGroup, clarifying how the Application will be constrained at the system level.

[SWS_EM_02102] 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] 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 Platform Health Management, which then starts recovery actions which have been configured beforehand. This will be the standard option for deterministic subsystems (see 7.6.1).
- For Processes without hard deadlines, resource violations sometimes can be mitigated without dedicated error recovery actions, e.g. by interrupting execution and continue at a later point in time.

- If the OS provides a way to limit resource consumption of a Process or a group of Processes by design, explicit external monitoring is usually not necessary and often not even possible. Instead, the limitation mechanisms make sure that resource availability for other parts of the system is not affected by failures within the enclosed Processes. When such by-design limitation is used, monitoring mechanisms may still be used for the benefit of the platform, but are not required. Self-monitoring and out-of-process monitoring is currently out-of-scope in 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 Policy

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

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

Refer to requirement [SWS_EM_01014] regarding Sheduling Policy configuration by Execution Management.

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.

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 must now be made available, the OS may have to execute some amounts of potentially-unbounded activities to make this memory available. Often, the reverse may also be happening, removing previously available (but unused) memory from the Process under scope, to make it available to other Processes. This is detrimental to an overall system determinism.

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] 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] 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] Process pre-mapping ⌈ Execution Management shall pre-map a Process if required by the corresponding Application Manifest. ⌋*(RS_EM_00005)*

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

7.8 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 [8].

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 `Adaptive Platform`.

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

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

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

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

7.8.3 Threat Model

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

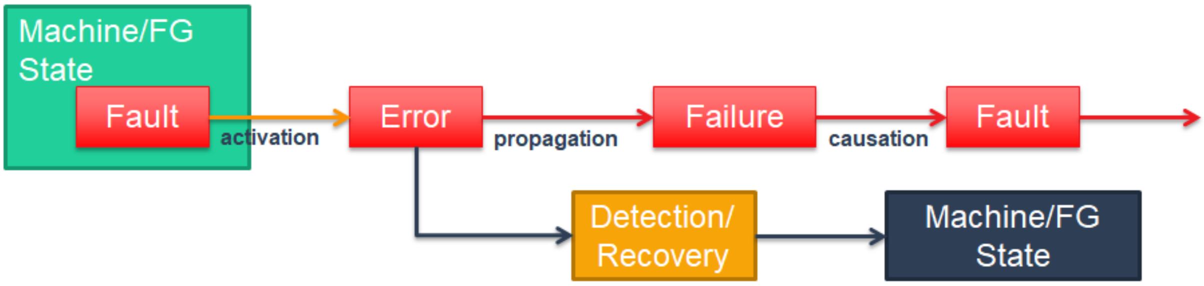

Figure 7.18: General Fault Tolerance scheme.

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 must be introduced to ensure error detection.

7.9 Handling of Application Manifest

7.9.1 Overview

The Application Manifest is created at design time by the Application Developer.

The Application Manifest specifies the deployment related information of an Executable running on the Adaptive Platform. An Application Manifest is bundled with the actual executable code in order to support the integration of the executable code onto the Machine.

For more information regarding the Application Manifest specification please see [3].

To perform its necessary actions, Execution Management imposes a number of requirements on the content of the Application Manifest. This section serves as a reference for those requirements.

7.9.2 Execution Dependency

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

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

7.9.3 Application Arguments

The set of static arguments required by a Process can either be provided by the Application developer or specified at integration time. The integrator then makes the arguments available in the Application Manifest for use by Execution Management when starting the Process [SWS_EM_01012].

7.9.4 Machine State and Function Group State

[SWS_EM_01013] Machine State and Function Group State ⌈ Execution Management shall support the execution of specific Processes depending on the current Machine State and Function Group States, based on information provided in the Application Manifests. ⌋*(RS_EM_00101)*

Each Process is assigned to one or several startup configurations (StartupConfig), which each can define the startup behaviour in one or several Machine States and/or Function Group States. For details see [3]. By parsing this information from the Application Manifests, Execution Management can determine which Processes need to be launched if a specific Machine State or Function Group State is entered, and which startup parameters are valid.

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

7.9.5 Scheduling Policy

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

For the detailed definitions of these policies, refer to [9]. Note, SCHED_OTHER shall be treated as non real-time scheduling policy, and actual behavior of the policy is implementation specific. It must not be assumed that the scheduling behavior is compatible between different Adaptive Platform implementations, except that it is a non 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)*

7.9.6 Scheduling Priority

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

The available priority range and actual meaning of the scheduling priority depends on the selected scheduling policy.

[SWS_EM_01039] Scheduling priority range for SCHED_FIFO and SCHED_RR ⌈ For SCHED_FIFO ([SWS_EM_01041]) and SCHED_RR ([SWS_EM_01042]), an integer between 1 (lowest priority) and 32 (highest priority) shall be used. ⌋*(RS_EM_00002)*

[SWS_EM_01040] Scheduling priority range for SCHED_OTHER ⌈ For the non real-time policy SCHED_OTHER ([SWS_EM_01043]) the scheduling priority shall always be zero. ⌋*(RS_EM_00002)*

7.9.7 Application Binary Name

The Application binary name (the name of the Executable) is included within the Application Manifest [TPS_MANI_01011]. Execution management can use the name to locate the Executable prior to starting the Process.

8 API specification

8.1 Type definitions

8.1.1 ApplicationState

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

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

Header file:	application_client.h		
Description:	Defines the states of an `Application` (see 7.4.2).		

Table 8.1: ApplicationState

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

8.1.2 ApplicationReturnType

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

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

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

Table 8.2: ApplicationReturnType

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

8.1.3 ActivationReturnType

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

Range:	`kRegisterServices`
	`kServiceDiscovery`
	`kInit`
		`0`
		`1`
		`2`
			`--`
			`--`
			`--`

	kRun kTerminate	3 4	-- --

Syntax:	`enum class ActivationReturnType : uint8_t {`
	`kRegisterServices = 0,`
	`kServiceDiscovery = 1,`
	`kInit = 2,`
	`kRun = 3,`
	`kTerminate = 4`
	`};`		

Header file:	deterministic_client.h		
Description:	Defines the return codes for WaitForNextActivation operations.		

Table 8.3: ActivationReturnType

[SWS_EM_02201] ActivationReturnType Enumeration ⌈Table 8.3 describes the enumeration `ActivationReturnType`.⌋*(RS_EM_00052)*

8.1.4 ActivationTimeStampReturnType

Name:	`ActivationTimeStampReturnType`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	kSuccess kNotAvailable	0 1	-- --

Syntax:	`enum class ActivationTimeStampReturnType : uint8_t {`
	`kSuccess = 0,`
	`kNotAvailable = 1`
	`};`		

Header file:	deterministic_client.h		
Description:	Defines the return codes for “get activation timestamp” operations.		

Table 8.4: ActivationTimeStampReturnType

[SWS_EM_02202] ActivationTimeStampReturnType Enumeration ⌈Table 8.4 describes the enumeration `ActivationTimeStampReturnType`.⌋*(RS_EM_00053)*

8.2 Class definitions

8.2.1 ApplicationClient class

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

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

8.2.1.1 ApplicationClient::ApplicationClient

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

Table 8.5: ApplicationClient::ApplicationClient

[SWS_EM_02030] ApplicationClient::ApplicationClient API ⌈Table 8.5 describes the interface `ApplicationClient::ApplicationClient`.⌋*(RS_EM_00103)*

8.2.1.2 ApplicationClient::~ApplicationClient

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

Table 8.6: ApplicationClient::~ApplicationClient

[SWS_EM_02002] ApplicationClient::~ApplicationClient API ⌈Table 8.6 describes the interface `ApplicationClient::~ApplicationClient`.⌋*(RS_EM_00103)*

8.2.1.3 ApplicationClient::ReportApplicationState

Service name:	ApplicationClient::ReportApplicationState	

Syntax:	`ApplicationReturnType ReportApplicationState(`
	`ApplicationState state`
	`);`	

Sync/Async:	Sync	
Parameters (in):	state	Value of the `Applications` state
Parameters (inout):	None	
Parameters (out):	None	

Return value:	kSuccess	Retrieval operation succeeded.
	kGeneralError	GeneralError
Exceptions:	None	
Description:	Interface for an Application to report the state to Execution Management.	

Table 8.7: ApplicationClient::ReportApplicationState

[SWS_EM_02003] ApplicationClient::ReportApplicationState API ⌈Table 8.7 describes the interface ApplicationClient::ReportApplicationState.⌋ *(RS_EM_00103)*

8.2.2 DeterministicClient class

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

[SWS_EM_02210] ⌈ The DeterministicClient class shall be declared in the deterministic_client.h header file. ⌋*(RS_EM_00052, RS_EM_00053)*

8.2.2.1 DeterministicClient::DeterministicClient

Service name:	DeterministicClient::DeterministicClient	
Syntax:	`DeterministicClient();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	Implementation specific	In case the underlying IPC mechanism fails.
Description:	Constructor for DeterministicClient which opens the Execution Managements communication channel (e.g. POSIX FIFO) to access a wait point for cyclic execution, a worker pool, deterministic random numbers and time stamps.	

Table 8.8: DeterministicClient::DeterministicClient

[SWS_EM_02211] DeterministicClient::DeterministicClient API ⌈Table 8.8 describes the interface DeterministicClient::DeterministicClient.⌋ *(RS_EM_00052, RS_EM_00053)*

8.2.2.2 DeterministicClient::~DeterministicClient

Service name:	DeterministicClient::~DeterministicClient
Syntax:	`~DeterministicClient();`
Sync/Async:	Sync
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	Destructor for DeterministicClient.

Table 8.9: DeterministicClient::~DeterministicClient

[SWS_EM_02215] DeterministicClient::~DeterministicClient API ⌈Table 8.9 describes the interface `DeterministicClient::~DeterministicClient`.⌋ *(RS_EM_00052, RS_EM_00053)*

8.2.2.3 DeterministicClient::WaitForNextActivation

Service name:	DeterministicClient::WaitForNextActivation	
Syntax:	`ActivationReturnType WaitForNextActivation ();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	`kRegisterServices`	application shall register communication services (this must be the only occasion for performing service registering).
	`kServiceDiscovery`	application shall do communication service discovery (this must be the only occasion for performing service discovery).
	`kInit`	application shall initialize its internal data structures (once).
	`kRun`	application shall perform its normal operation.
	`kTerminate`	application shall terminate
Exceptions:	None	
Description:	Blocks and returns with a process control value when the next activation is triggered by the Runtime.	

Table 8.10: DeterministicClient::WaitForNextActivation

[SWS_EM_02216] DeterministicClient::WaitForNextActivation API ⌈Table 8.10 describes the interface `DeterministicClient::WaitForNextActivation`.⌋ *(RS_EM_00052)*

8.2.2.4 DeterministicClient::RunWorkerPool

Service name:	DeterministicClient::RunWorkerPool

Syntax:	`void RunWorkerPool (`
	`Worker &runnableObj,`
	`Container &container`
	`);`	

Sync/Async:	Sync	
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++
Parameters (inout):	None	
Parameters (out):	None	
Return value:	void	
Exceptions:	None	
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.	

Table 8.11: DeterministicClient::RunWorkerPool

[SWS_EM_02220] DeterministicClient::RunWorkerPool API ⌈Table 8.11 describes the interface `DeterministicClient::RunWorkerPool`.⌋*(RS_EM_00053)*

8.2.2.5 DeterministicClient::GetRandom

Service name:	DeterministicClient::GetRandom	
Syntax:	`uint64_t GetRandom ();`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	uint64_t	64 bit uniform distributed pseudo random number
Exceptions:	None	
Description:	This returns “Deterministic” random numbers. ‘Deterministic” means, that the returned random numbers are identical within redundant `DeterministicClient::WaitForNextActivation`() cycles, which are used within redundantly executed `Processes`.	

Table 8.12: DeterministicClient::GetRandom

[SWS_EM_02225] DeterministicClient::GetRandom API ⌈Table 8.12 describes the interface `DeterministicClient::GetRandom`.⌋*(RS_EM_00053)*

8.2.2.6 DeterministicClient::GetActivationTime

Service name:	DeterministicClient::GetActivationTime	
Syntax:	`ActivationTimeStampReturnType GetActivationTime (TimeStamp);`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	std::chrono:: time_point <Synchronized Time Base, Duration>&	current activation time
Return value:	`kSuccess`	Operation successful
	`kNotAvailable`	No previous call of WaitForNextActivation with return value kRun
Exceptions:	None	
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`.	

Table 8.13: DeterministicClient::GetActivationTime

[SWS_EM_02230] DeterministicClient::GetActivationTime API ⌈Table 8.13 describes the interface `DeterministicClient::GetActivationTime`.⌋ *(RS_EM_00053)*

8.2.2.7 DeterministicClient::GetNextActivationTime

Service name:	DeterministicClient::GetNextActivationTime	
Syntax:	`ActivationTimeStampReturnType GetNextActivationTime (TimeStamp);`	
Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	std::chrono:: time_point <Synchronized Time Base, Duration>&	next activation time
Return value:	`kSuccess`	Operation successful
	`kNotAvailable`	Next activation time unknown
Exceptions:	None	
Description:	This provides the timestamp that represents the point in time when the next activation will be triggered by `DeterministicClient::WaitForNextActivation`() with return value kRun. Subsequent calls within an activation cycle will always provide the same value. The same value will also be provided within redundantly executed `Processes`.	

Table 8.14: DeterministicClient::GetNextActivationTime

[SWS_EM_02235] DeterministicClient::GetNextActivationTime API ⌈Table 8.14 describes the interface `DeterministicClient::GetNextActivationTime`.⌋ *(RS_EM_00053)*

9 Service Interfaces

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

9.1 Service Type definitions

9.1.1 StateStatusType

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

9.2 State Management Interface

9.2.1 Methods

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

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

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

9.2.2 Events

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

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

A Not applicable requirements

[SWS_EM_NA] ⌈ These requirements are not applicable as they are not within the scope of this release. ⌋*(RS_EM_00003, RS_EM_00004, RS_EM_00050, RS_EM_00051, RS_EM_00110)*

B Mentioned Class Tables

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

Enumeration	**CommandLineOptionKindEnum**
Package	M2::AUTOSARTemplates::AdaptivePlatform::Deployment::Process
Note	This enum defines the different styles how the command line option appear in the command line. **Tags:** atp.Status=draft
Literal	***Description***
commandLineLongForm	Long form of command line option. Example: --version=1.0 --help **Tags:** atp.EnumerationValue=1
commandLineShortForm	Short form of command line option. Example: -v 1.0 -h **Tags:** atp.EnumerationValue=0
commandLineSimpleForm	In this case the command line option does not have any formal structure. Just the value is passed to the program. **Tags:** atp.EnumerationValue=2

Table B.1: CommandLineOptionKindEnum

Class	**Executable**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure			
Note	This meta-class represents an executable program. **Tags:** atp.Status=draft; atp.recommendedPackage=Executables			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
buildType	BuildTypeEnum	0..1	attr	This attribute describes the buildType of a module and/or platform implementation.
minimumTimerGranularity	TimeValue	0..1	attr	This attribute describes the minimum timer resolution (TimeValue of one tick) that is required by the Executable. **Tags:** atp.Status=draft
rootSwComponentPrototype	RootSwComponentPrototype	0..1	aggr	This represents the root SwCompositionPrototype of the Executable. This aggregation is required (in contrast to a direct reference of a SwComponentType) in order to support the definition of instanceRefs in Executable context. **Tags:** atp.Status=draft
transformationPropsMappingSet	TransformationPropsToServiceInterfaceElementMappingSet	0..1	ref	Reference to a set of serialization properties that are defined for ServiceInterfaces of the Executable. **Tags:** atp.Status=draft
version	String	0..1	attr	Version of the executable. **Tags:** atp.Status=draft

Table B.2: Executable

Class	**ModeDeclaration**			
Package	M2::AUTOSARTemplates::CommonStructure::ModeDeclaration			
Note	Declaration of one Mode. The name and semantics of a specific mode is not defined in the meta-model. **Tags:** atp.ManifestKind=ApplicationManifest,MachineManifest			
Base	*ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
value	PositiveInteger	0..1	attr	The RTE shall take the value of this attribute for generating the source code representation of this ModeDeclaration.

Table B.3: ModeDeclaration

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

Table B.4: ModeDependentStartupConfig

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Deployment::Process			
Note	This meta-class provides information required to execute the referenced executable. **Tags:** atp.ManifestKind=ApplicationManifest; atp.Status=draft; atp.recommendedPackage=Processes			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
applicationModeMachine	ModeDeclarationGroupPrototype	0..1	aggr	Set of ApplicationStates (Modes) that are defined for the process. **Tags:** atp.Status=draft
design	ProcessDesign	0..1	ref	This reference represents the identification of the design-time representation for the Process that owns the reference. **Tags:** atp.Status=draft

executable	Executable	0..1	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef **Tags:** atp.Status=draft
logTraceDefaultLogLevel	LogTraceDefaultLogLevelEnum	0..1	attr	This attribute allows to set the initial log reporting level for a logTraceProcessId (ApplicationId).
logTraceFilePath	UriString	0..1	attr	This attribute defines the destination file to which the logging information is passed.
logTraceLogMode	LogTraceLogModeEnum	0..1	attr	This attribute defines the destination of log messages provided by the process.
logTraceProcessDesc	String	0..1	attr	This attribute can be used to describe the logTraceProcessId that is used in the log and trace message in more detail.
logTraceProcessId	String	0..1	attr	This attribute identifies the process in the log and trace message (ApplicationId).
modeDependentStartupConfig	ModeDependentStartupConfig	*	aggr	Applicable startup configurations. **Tags:** atp.Status=draft

Table B.5: Process

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

Table B.6: Referrable

Class	**StartupConfig**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Deployment::Process			
Note	This meta-class represents a reusable startup configuration for processes.. **Tags:** atp.ManifestKind=ApplicationManifest; atp.Status=draft			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
schedulingPolicy	SchedulingPolicyKindEnum	0..1	attr	This attribute represents the ability to define the scheduling policy for the initial thread of the application.
schedulingPriority	Integer	0..1	attr	This is the scheduling priority requested by the application itself.
startupOption	StartupOption	*	aggr	Applicable startup options **Tags:** atp.Status=draft

Table B.7: StartupConfig

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

Table B.8: StartupOption

C Interfaces to other Functional Clusters (informative)

C.1 Overview

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

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

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

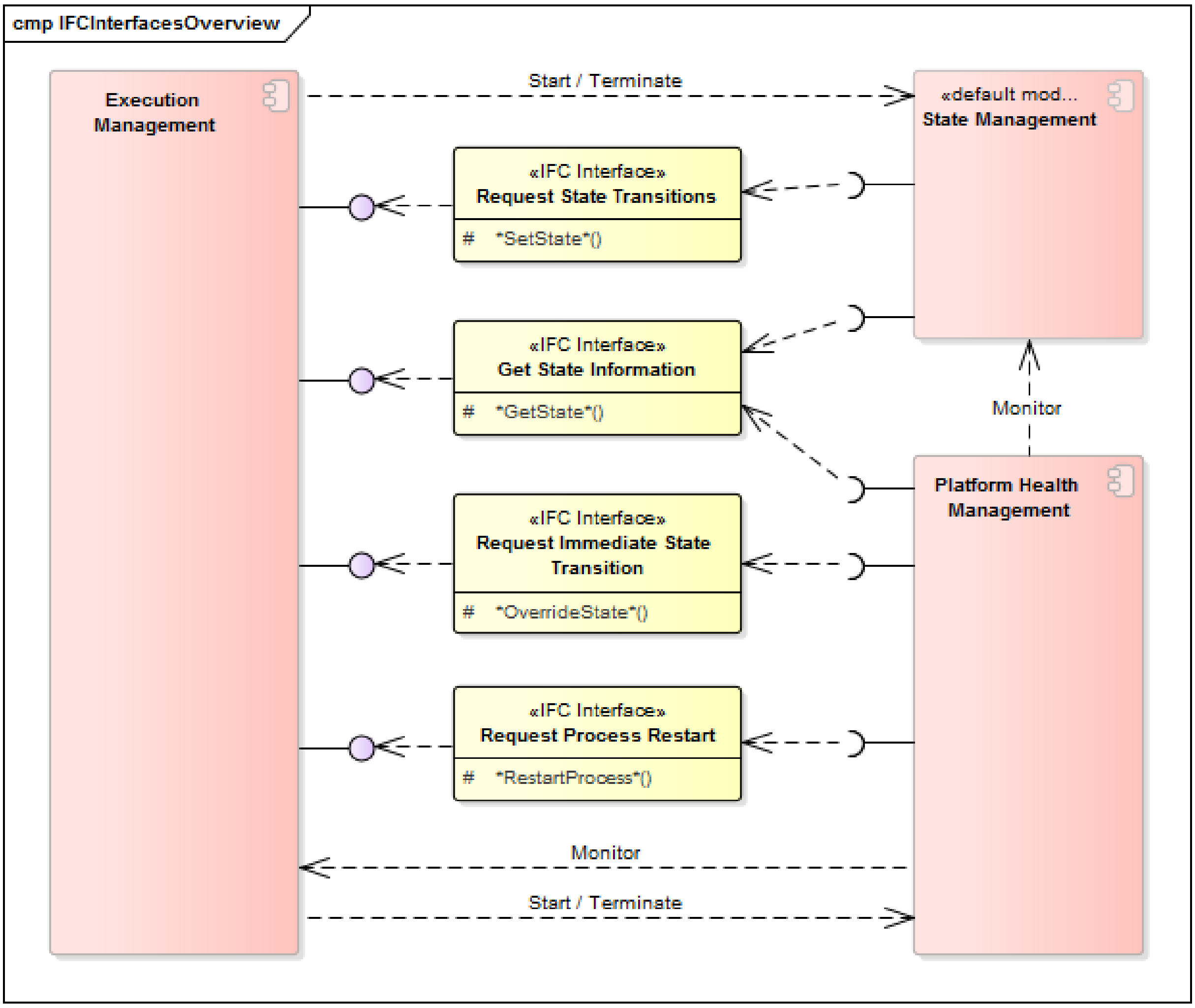

Figure C.1: Interfaces between Functional Clusters

C.2 Interface Tables

C.2.1 State Transition Request

	Name	Description	Requirements
Intended users	State Management		
Name proposal	*SetState*		

Functionality	Requests a change of Function Group States and/or Machine States	The state change request shall lead to one or several state transitions and hereof state changes to the requested Machine State and/or Function Group States	[SWS_EM_01026] [SWS_EM_01060] [SWS_EM_01065] [SWS_EM_01066] [SWS_EM_01067] [SWS_EM_01068]
Parameters (in)	Function Group	Identifier of Function Group (as defined in Machine Manifest) or "MachineState" to request a Machine State.	[SWS_EM_01107]
	State	Requested state of the Function Group or Machine State. States are defined in the Machine Manifest. 1..* pairs of <Function Group><State> can be requested atomically.	[SWS_EM_01032] [SWS_EM_01108]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_02057]
	Execution Management is busy and cannot accept request	State change requests, that are received before all previously requested Machine State and/or Function Group State transitions are completed	[SWS_EM_01034]
	State change request could not be finished in time	Timeout detected at state transition	[SWS_EM_02058]
	general error		[SWS_EM_02056]

Table C.1: State Transition Request

C.2.2 State Override Request

	Name	Description	Requirements
Intended users	Platform Health Management		
Name proposal	*OverrideState*		
Functionality	Requests a change of Function Group States and/or Machine States and stops any currently "ongoing" state changes	The state change request shall immediately lead to one or several state transitions and hereof state changes to the requested Machine State and/or Function Group States	[SWS_EM_01018] [SWS_EM_01061]
Parameters (in)	Function Group	Identifier of Function Group (as defined in Machine Manifest) or "MachineState" to request a Machine State.	[SWS_EM_01107]

	State	Requested state of the Function Group or Machine State. States are defined in the Machine Manifest. 1..* pairs of <Function Group><State> can be requested atomically.	[SWS_EM_01032] [SWS_EM_01108]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_02057]
	State change request could not be finished in time	Timeout detected at state transition	[SWS_EM_02058]
	general error		[SWS_EM_02056]

Table C.2: State Override Request

C.2.3 Provide State Information

	Name	Description	Requirements
Intended users	State Management		
	Platform Health Management		
Name proposal	*GetState*		
Functionality	Get information about current state	The Execution Management provides an interface to retrieve the current Machine State or a Function Group State.	[SWS_EM_01028]
Parameters (in)	Function Group	Identifier of Function Group (as defined in Machine Manifest) or “MachineState” to request a Machine State.	[SWS_EM_01107]
Parameters (inout)	None		
Parameters (out)	State	Current Function Group State of the given Function Group or the current Machine State of the given Function Group name "MachineState". Empty if retrieval operation was not successful.	[SWS_EM_01032] [SWS_EM_01108]
Return value	Operation succeeded		[SWS_EM_02050]
	Execution Management is busy and cannot provide requested information	Execution Management performs a State transition of the requested Function Group or Machine State	[SWS_EM_02044]
	general error	A state transition of the requested Function Group or Machine State failed	[SWS_EM_02049]

Table C.3: Provide State Information

C.2.4 Process Restart Request

	Name	Description	Requirements

Intended users	Platform Health Management		
Name proposal	*RestartProcess*		
Functionality	Request to restart a process	Restart a specific process on the request from the Platform Health Management.	[SWS_EM_01016] [SWS_EM_01062]
Parameters (in)	process identifier	Unique identifier of the process to be restarted.	[SWS_EM_01016]
Parameters (inout)	None		
Parameters (out)	None		
Return value	Operation succeeded		[SWS_EM_01064]
	general error	process could not be restarted	[SWS_EM_01063]

Table C.4: Process Restart Request