Document Title	Specification of State Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	908

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-10

Document Change History

Date	Release	Changed by	Description
2018-10-31	18-10	AUTOSAR Release Management	• Initial release

Disclaimer

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

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

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

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

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction and functional overview

This document is the software specification of the State Management functional cluster within the Adaptive Platform Services.

State Management is responsible for determination of the overall operation state of the Adaptive AUTOSAR Platform. Additionally State Management takes care to set Components into an internal Component State which depends on the determined Operational State and on other project specific requirements. The idea behind Component States is to control Processs in a more fine-grained way without the need of restarting them with a different set of command-line parameters.

State Management interacts with the Execution Management to request Function Groups and the Machine State to enter specific states that are determined to project requirements.

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

Chapter 8 documents the State Management Application Programming Interface (API).

1.1 What is State Management?

State Management is the functional cluster within the Adaptive Platform Services that is responsible to determine the operation state, based on information received or gathered from other Adaptive Platform Applications or Adaptive Applications.

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 State Management specified in Chapter 8 . Note that APIs accessed by State Management using the inter-functional cluster API that is described in Appendix A which is not part of ARA.

The Adaptive AUTOSAR Services are provided via mechanisms provided by the Communication Management functional cluster [1] of the Adaptive Platform Foundation

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the State Management module that are not included in the AUTOSAR glossary[2].

Terms:	Description:
State Management	The element defining modes of operation for AUTOSAR Adaptive Platform. It allows flexible definition of functions which are active on the platform at any given time.
Execution Management [3]	A Functional Cluster within the Adaptive Platform Foundation
Communication Management [1]	A Functional Cluster within the Adaptive Platform Foundation
Network Management [4]	A Functional Cluster within the Adaptive Platform Services. Part of Communication Management.
Adaptive Diagnostics [5]	A Functional Cluster within the Adaptive Platform Services
Update And Config Management [6]	A Functional Cluster within the Adaptive Platform Services
Network Handle	Network Handles are provided by Network Management. A handle represents a set of (partial) networks.
Process	A process is a loaded instance of an Executable to be executed on a Machine.
Function Group	A Function Group is a set of coherent Processes, which need to be controlled consistently. Depending on the state of the Function Group, Processes are started or terminated.
Component	Element of a Process. Processes are comprised of one or more SW-entities that provide a particular function or group of related functions called Component. **Please note that the term 'Component' is not yet fixed for this scope.**
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 configured in the Machine Manifest [7].
Machine State	The state of Function Group "MachineState" with some predefined states (Startup/Shutdown/Restart).
Operational State	The element of State Management that characterizes the current internal state of the State Management. The Operational State is machine specific and depends on multiple events from somewhere within the system.
Component State	The element of State Management that characterizes the current state of Components within an Adaptive Application. The Component State is Adaptive Application specific and therefore it has to be described in the respective Execution Manifest. Every Adaptive Application provides at least an "On" and an "Off" State.
Execution Manifest	Manifest file to configure execution of an Adaptive Application.
Machine Manifest	Manifest file to configure a Machine.

Table 2.1: Acronyms and Abbreviations

The following technical terms used throughout this document are defined in the official [2] AUTOSAR Glossary or [7] TPS Manifest Specification – they are repeated here for tracing purposes.

Term	Description
Adaptive Application	see [2] AUTOSAR Glossary
Application	see [2] AUTOSAR Glossary
AUTOSAR Adaptive Platform	see [2] AUTOSAR Glossary
Adaptive Platform Foundation	see [2] AUTOSAR Glossary
Adaptive Platform Services	see [2] AUTOSAR Glossary
Manifest	see [2] AUTOSAR Glossary
Executable	see [2] AUTOSAR Glossary
Functional Cluster	see [2] AUTOSAR Glossary
Machine	see [2] AUTOSAR Glossary
Service	see [2] AUTOSAR Glossary
Service Interface	see [2] AUTOSAR Glossary
Service Discovery	see [2] AUTOSAR Glossary

Table 2.2: Glossary-defined Technical Terms

3 Related documentation

3.1 Input documents & related standards and norms

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

[1] Specification of Communication Management
AUTOSAR_SWS_CommunicationManagement

[2] Glossary
AUTOSAR_TR_Glossary

[3] Specification of Execution Management
AUTOSAR_SWS_ExecutionManagement

[4] Specification for Network Management
AUTOSAR_SWS_NetworkManagement

[5] Specification of Diagnostics
AUTOSAR_SWS_Diagnostics

[6] Specification of Update and Configuration Management
AUTOSAR_SWS_UpdateAndConfigManagement

[7] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[8] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral

[9] Requirements on Persistency
AUTOSAR_RS_Persistency

[10] Requirements of State Management
AUTOSAR_RS_StateManagement

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [8, SWS BSW General], which is also valid for State Management.

Thus, the specification SWS BSW General shall be considered as additional and required specification for StateManagement

4 Constraints and assumptions

4.1 Limitations

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

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

- Section 7.3 Component States are partially discussed. Information which should be available from Execution Manifest to enable Component functionality is not yet modeled. Model information for Components and Component States will be available in the next release.
- Section 7.4 Communication Control for Diagnostic reasons this is not yet discussed with Adaptive Diagnostics. It will be expanded in the next release.
- Section 7.4 RequestRestart for Diagnostic reasons this is discussed with Adaptive Diagnostics, but some interface details are not yet finalized. It will be expanded in a next release.

4.2 Applicability to car domains

If a superior State Management instance to the one from the ECU is available in a hierarchical car context, the State Management of the ECU shall also evaluate events generated by the superior instance of State Management. Section 7.8 will give further details.

5 Dependencies to other modules

5.1 Platform dependencies

5.1.1 Operating System Interface

State Management has no direct interface to the Operating System. All OS dependencies are abstracted by the Execution Management and Persistency.

5.1.2 Execution Manager Interface

State Management is dependent on Execution Management to start and stop processes - as part of the defined Function Groups or Machine States.

5.1.3 Persistency

State Management is dependent on the Persistency [9] functional cluster. Persistency is used to access persistent storage.

5.1.4 Adaptive Diagnostics

State Management is dependent on the Adaptive Diagnostics [5] functional cluster. Adaptive Diagnostics provides information about an ongoing diagnostics session. This information is evaluated by State Management to prevent shutdown of the system when a diagnostics session is ongoing.

5.1.5 Update And Config Management

State Management is dependent on the Update and Config Management [6] functional cluster. Update and Config Management provides information about an ongoing update session. This information is evaluated by State Management to prevent shutdown of the system when an update session is ongoing.

5.1.6 Network Management

State Management is dependent on the Network Management [4] functional cluster. Network Management provides multiple NetworkHandle fields which represents a set of (partial) networks. State Management evaluates this fields to set Function Groups to the corresponding Function Group State defined in Manifest and vice versa. Additionally State Management shall prevent system from shutting down during an update session is ongoing.

5.2 Other dependencies

Currently, there are no other library dependencies.

6 Requirements Tracing

The following tables reference the requirements specified in [10] 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_SM_00001]	State Management shall support Function Group state change requests.	[SWS_SM_0001] [SWS_SM_0002] [SWS_SM_0003] [SWS_SM_0004] [SWS_SM_0005] [SWS_SM_0006] [SWS_SM_0400] [SWS_SM_0401] [SWS_SM_0402]
[RS_SM_00002]	State Management shall support Component State change requests.	[SWS_SM_0010] [SWS_SM_0011] [SWS_SM_0012] [SWS_SM_0013] [SWS_SM_0014] [SWS_SM_0015] [SWS_SM_0102]
[RS_SM_00100]	State Management shall support ECU reset	[SWS_SM_0100] [SWS_SM_0101] [SWS_SM_0102] [SWS_SM_0103] [SWS_SM_0104] [SWS_SM_0105] [SWS_SM_0200] [SWS_SM_0201]
[RS_SM_00101]	State Management shall support diagnostic reset cause	[SWS_SM_0103] [SWS_SM_0104] [SWS_SM_0105]
[RS_SM_00200]	State Management shall provide an interface between State Management instances.	[SWS_SM_0500] [SWS_SM_0501]
[RS_SM_00201]	State Management shall provide the interface over ara::com.	[SWS_SM_0500] [SWS_SM_0501]
[RS_SM_00300]	State Management shall support variant handling based on calibration data.	[SWS_SM_0005] [SWS_SM_0006]
[RS_SM_00400]	State Management shall establish communication paths dynamically.	[SWS_SM_0300] [SWS_SM_0301] [SWS_SM_0302] [SWS_SM_0303] [SWS_SM_0304]

7 Functional specification

State Management is a functional cluster contained in the Adaptive Platform Services. State Management is responsible for all aspects of Operation State Management including handling of incoming events or requests, prioritization of these events/requests and setting the Operational State. The State Management can also register to properties of other Adaptive Platform or Adaptive Applications to be informed about changes. These changes can also trigger a transition in the Operational State machine.

Additionally the State Management takes care of not shutting down the system as long as any diagnostic or update session is active as part of its internal Operational State.

In dependency of the current Operational States, State Management might decide to request Function Groups or Machine State to enter specific state by using interfaces of Execution Management.

State Management is responsible for en- and disabling (partial) networks by means of Network Management. Network Management provides ara::com fields (NetworkHandle) where each of the fields represents a set of (partial) networks. State Management can influence these fields in dependency of Function Groups states and - vice versa - can set Function Groups to a defined state depending on the value of Network Managements NetworkHandle fields.

From the State Management internal Operational States more fine grained Adaptive Application and AUTOSAR Adaptive Platform Application internal states can be derived. They are called Component States.

This chapter describes the functional behaviour of State Management and the relation to other Adaptive Platform Applications State Management interacts with.

- Section 7.1 presents an introduction to key terms within State Management focusing on the relationship between Processes, Components, Operational States and Component States.
- Section 7.2 covers the core State Management run-time responsibilities including the start of Applications.
- Section 7.3 describes what Components and Component States are and how they are used.
- Section 7.4 covers several topics related to Adaptive Diagnostics including shutdown prevention and executing of different reset types
- Section 7.5 describes how Update and Config Management interacts with State Management

- Section 7.6 documents support provided by Network Management to de-/activate (partial) networks in dependency of Function Group States and vice versa.
- Section 7.7 describes how Execution Management is used to change Function Group State or Machine State.
- Section 7.8 provides an introduction to how State Management will work within a virtualized environment.

7.1 Technical Overview

This chapter presents a short summary of the relationship between State Management, and all AUTOSAR Adaptive Platform Applications which are mentioned in the dependency section 5 of this document.

7.1.1 Terms

Before discussing the concepts of Operational State, Components, and Component States it is useful to present an overview of the terms so that the more detailed discussions have the required context.

Operational State – An internal state within State Management derived from multiple events in the system. TheOperational State may be used to determine other states in the State Management e.g. Component States and Function Group States

Component – A Process is comprised of one or more Components. Components are SW-entities that provide a particular function or group of related functions.

Component State – Component State is the internal state of Components. It is used to enable Applications to be controlled in a more fine-grained way without the need of restarting Applications with a different parameter set. Component States are derived from Operational State or other project specific requirements by State Managements internal logic. At least ON and OFF state shall be supported.

ON – full functionality provided.

OFF – no external functionality provided, all persisted data stored, ready to be terminated.

7.2 State Management Responsibilities

State Management is the functional cluster which is responsible for determining the current Operational State, and for initiating Function Group and Machine State transitions by requesting them from Execution Management.

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.

State Management is the central point where any operation event is received that might have an influence to the Operational State. The State Management is responsible to evaluate these events and decide based on

- Event type (defined in project specific implementation based on project specific requirements).
- Event priority (defined in project specific implementation based on project specific requirements).
- Application identifier (Application identifier is not supported in this release. It is under discussion with FT-SEC if such an identifier could be provided by Identity and Access Management).

If an Operational State change is triggered Execution Management may be requested to set Function Groups or Machine State into new States.

The state change request for Function Groups can be issued by several AUTOSAR Applications, e.g.:

- Platform Health Management to trigger error recovery, e.g. to activate fallback Functionality
- Adaptive Diagnostics, to switch the system into diagnostic states
- Update and Config 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

AUTOSAR Applications may provide their own property or event via an ara com interface, where the State Management is subscribing to, to trigger State Management internal events. Since State Management functionality is critical, access from other AUTOSAR Applications must be secured, e.g. by Identity and Access Management.

- State Management may be monitored and supervised by Platform Health Management.

- State Management provides interfaces to request information about current states

State Management is responsible for handling the following states:

- Machine State see 7.2.1
- Function Group State see 7.2.2

7.2.1 Machine State

A Machine State is a specific type of Function Group State (see 7.2.2). Machine States and all other Function Group States are determined and requested by the State Management functional cluster, see 7.2.3. The set of active States is significantly influenced by vehicle-wide events and modes which are evaluated into State Managements internal Operational States.

The Function Group States, including the Machine State, define the current set of running Processes. Each Application can declare in its Execution Manifests in which Function Group States its Processes have to be running.

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 7.1 as an example Driving State is no mandatory State.

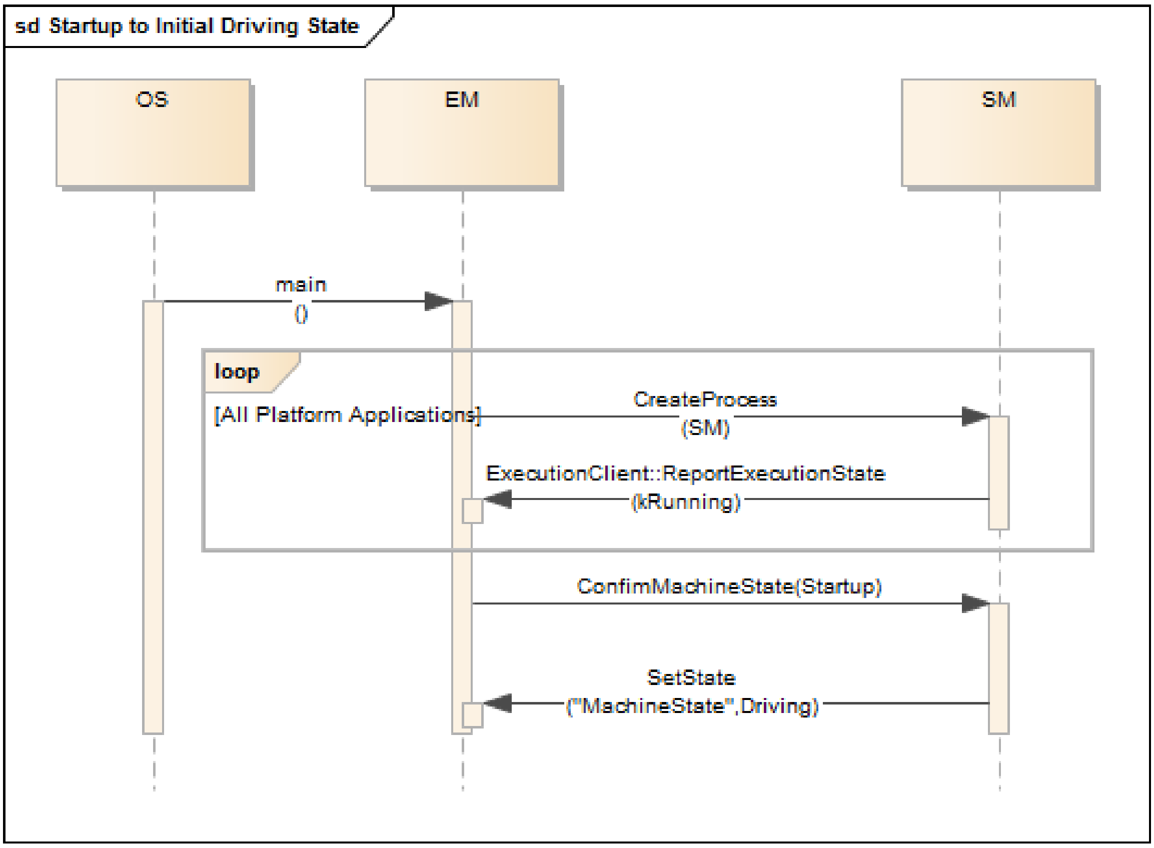
sd Startup to Initial Driving State
OS
EM
SM
main
()
loop
[All Platform Applications]
CreateProcess
(SM)
ExecutionClient::ReportExecutionState
(kRunning)
ConfimMachineState(Startup)
SetState
("MachineState",Driving)

Figure 7.1: Start-up Sequence – from Startup to initial running state Driving

An arbitrary state change sequence to machine state StateXYZ is illustrated in Figure 7.2. Here, on receipt of the state change request, Execution Management terminates running Processes and then starts Processes active in the new state before confirming the state change to State Management.

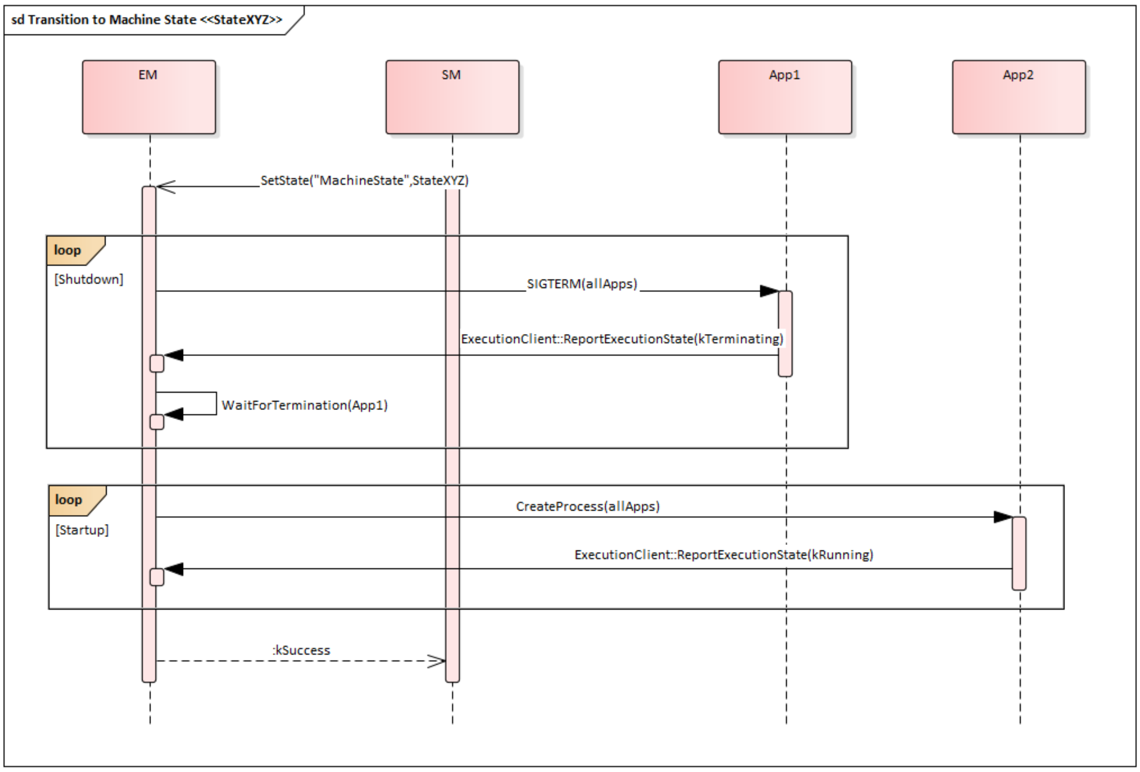
sd Transition to Machine State <<StateXYZ>>
EM
SM
App1
App2
SetState("MachineState",StateXYZ)
loop
[Shutdown]
SIGTERM(allApps)
ExecutionClient::ReportExecutionState(kTerminating)
WaitForTermination(App1)
loop
[Startup]
CreateProcess(allApps)
ExecutionClient::ReportExecutionState(kRunning)
:kSuccess

Figure 7.2: State Change Sequence – Transition to machine state StateXYZ

7.2.1.1 Startup

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

7.2.1.2 Shutdown

As mentioned in 7.2.1.1 AUTOSAR assumes that State Management will be configured to run in Shutdown. State transition is not a trivial system change and it can fail for a number of reasons. When ever this happens you may want State Management to be still alive, so you can report an error and wait for further instructions. Please note that very purpose of this state is to shutdown Machine (this includes State Management) in a clean manner. Unfortunately this means that at some point State Management will no longer be available and it will not be able to report errors anymore. Those errors will be handled in a implementation specific way.

7.2.1.3 Restart

As mentioned in 7.2.1.1 AUTOSAR assumes that State Management will be configured to run in Restart. The reasons for doing so are the same as for 7.2.1.2.

7.2.2 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, additional Function Groups and Function Group States can be configured. Other use cases where starting and terminating individual groups of Processes might be necessary including diagnostice and error recovery.

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

[SWS_SM_0001] Available Function Group (states) ⌈ State Management shall obtain available Function Groups and their potential states from the Machine Manifest to set-up the Function Group specific state management. ⌋ *(RS_SM_00001)*

Processes reference in their Execution Manifest the states in which they want to be executed. A state can be any Function Group State, including a Machine State. For details see [7], especially "Mode-dependent Startup Configuration" chapter and "Function Groups" chapter.

The arbitrary state change sequence as shown in Figure 7.2 applies to state changes of any Function Group - just replace "MachineState" by the name of the Function Group. On receipt of the state change request, Execution Management terminates not longer needed Processes and then starts Processes active in the new Function Group State before confirming the state change to State Management.

[SWS_SM_0002] Function Group State Change Request ⌈ State Management shall implement functionality to enable Adaptive Applications and Adaptive Platform Applications to change the Function Group State of Function Groups ⌋*(RS_SM_00001)*

It might be that State Management declines or delays the request to change a Function Groups state, based on State Management internal Operational State or another Adaptive Application or Adaptive Platform Application with higher priority that has the ownership of a Function Group. As per current specification several Adaptive Applications or Adaptive Platform Applications use the service interface of State Management e.g. Update and Config Management and a superior Function Group Manager. To ensure that the decision to set Function Groups into a dedicated Function Group State of a ”more important” application is not ”undermined” by a ”less important” application, the application with a higher priority (project specific) get the ownership of the requested Function Groups as long as it does not release the request.

[SWS_SM_0003] Function Group State Retrieval ⌈ State Management shall implement functionality to enable Adaptive Applications and Adaptive Platform Applications to retrieve the Function Group State of Function Groups and State of Machine State ⌋*(RS_SM_00001)*

[SWS_SM_0004] Function Group State Change Request Result ⌈ State Management shall return an appropriate result to the Adaptive Applications and Adaptive Platform Applications which has requested a Function Group State change. ⌋*(RS_SM_00001)*

The system might contain calibration data for variant handling. This might include that some of the Function Groups configured in the Machine Manifest are not intended to be executed on this system. therefore State Management has to evaluate calibration data to gather information about Function Groups not configured for the system variant

[SWS_SM_0005] Function Group Calibration Support ⌈ State Management shall receive information about deactivated Function Groups from calibration data. ⌋*(RS_SM_00001, RS_SM_00300)*

The storage and reception of calibration data is implementation specific.

[SWS_SM_0006] Function Group Calibration Support ⌈ State Management shall decline the request of Adaptive Applications and Adaptive Platform Applications to change the Function Group State of a Function Group which is not configured to run in this variant. ⌋*(RS_SM_00001, RS_SM_00300)*

7.2.3 State Management Architecture

State Management is the functional cluster which is responsible for determining the current set of active Function Group States, including the Machine State, 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 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.

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 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.

An overview of the interaction of State Management, Execution Management and Applications is shown in Figure 7.3.

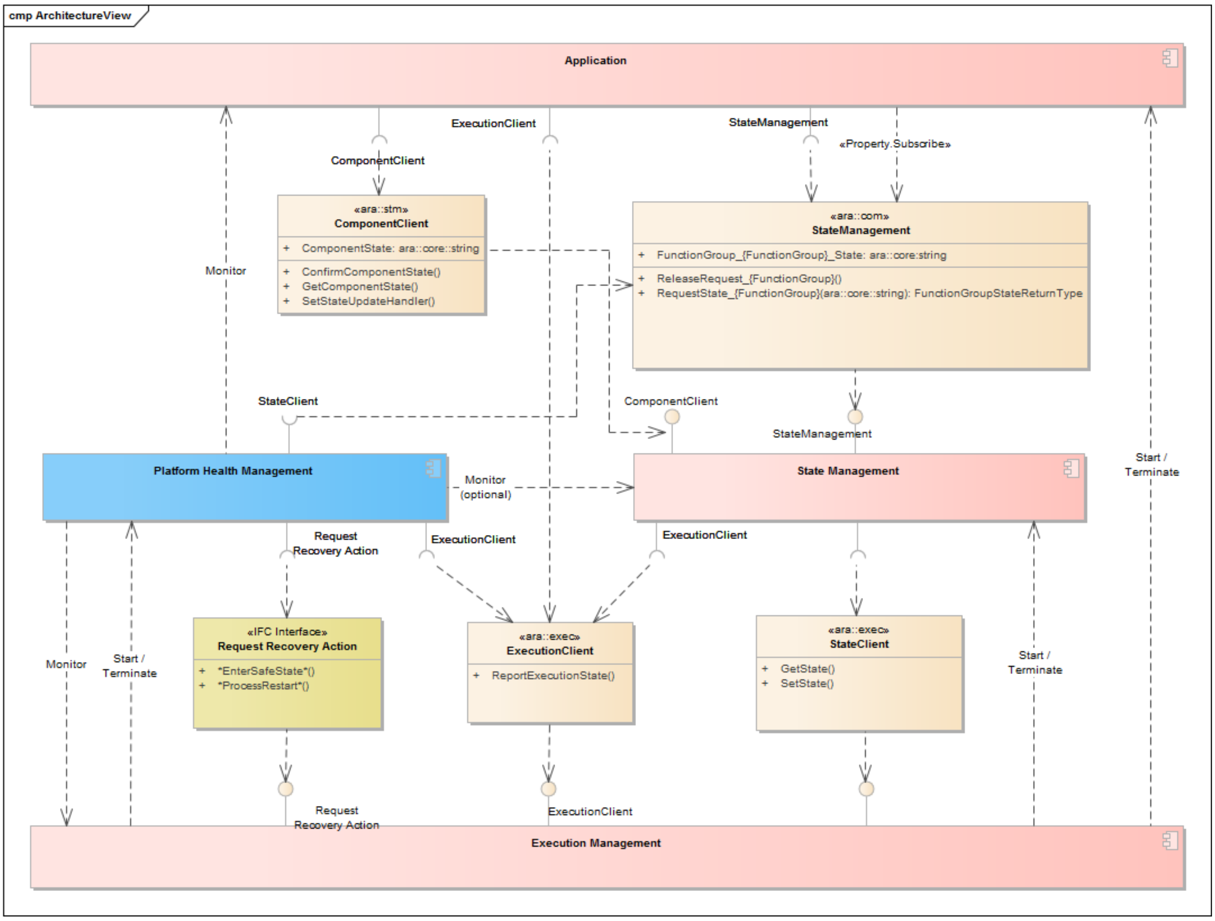

Figure 7.3: State Management Architecture

7.3 State Management and Components

Please note that the term 'Component' is not yet finally decided and therefore subject of change!

A single process is comprised of one or multiple Components. A component is e.g. a thread. To fulfill the needs of a resource optimized system it is necessary to control Processs and therefore their Components in a more fine-grained way than it is possible by Execution Management. When the internal behavior of a Process should be changed by Execution Management it is needed to unload Process from memory (including high latency due to persisting) and reload the Executable from filesystem to memory. This behavior is resource consuming with respect to (flash-)memory bandwidth, CPU load and execution time.

So therefore Components and their corresponding states are introduced. The Component States are derived by State Management from internal Operational States and from project specific requirements.

One important use-case is the 'late-wakeup', where a new wakeup reason is found during a running shutdown. With the current approach the shutdown can't be interrupted and all Processs have to be unloaded and newly loaded. With the Component States it is possible to switch all Components to their 'OFF' state, where all e.g. persisting should be done (like when a Process gets termination request from Execution Management), but they will stay in memory and can continue their work immediately when they are set into Component State 'ON' again by State Management.

[SWS_SM_0010] Component (states) ⌈ State Management shall enable Processes to change their internal behavior without the need of being reloaded. ⌋ *(RS_SM_00002)*

[SWS_SM_0011] Component (states) Handling ⌈ State Management shall calculate Component States from current Operational State and other project specific requirements and send the state to the registered Components. ⌋ *(RS_SM_00002)*

To enable Components to receive Component States they have to register at State Manager via its API interface (see section 8.2 API function definition). When Components are not longer interested in receiving Component States they have to un-register form State Manager, thus these Components are removed from State Managers internal list. Registration is done by calling the consturctor of Component-Client, un-registration is done by calling its destructor.

[SWS_SM_0012] Component (states) Registration ⌈ State Management shall provide means to Components to register / un-register for receiving Component States. ⌋*(RS_SM_00002)*

Components are allowed to temporary delay the next provided Component State when it sees a reason to do so (e.g. an OFF state might be delayed due to an ongoing phone call). therefore each Component has to confirm that it has received the request to enter a new Component State. This confirmation shall contain the current state (either the requested one or the previous one) of the Component and a result. For details see section 8.2.1.5. When the transition to the requested state is delayed State Management retries to request the Component State after a configured timeout has exceeded. This retry shall be done a configured number of times. Timeout values and retry counts shall be retrieved from Execution Manifest

[SWS_SM_0013] Component (states) Configuration ⌈ State Management shall retrieve configuration parameters for Components from Execution Manifest. ⌋ *(RS_SM_00002)*

[SWS_SM_0014] Component (states) Enforcement ⌈ State Management shall force to enter a Component State when the configured retry count and timeout values are exceeded. ⌋*(RS_SM_00002)*

[SWS_SM_0015] Component (states) Transitions ⌈ Components must be able to perform a transition from any Component State into any Component State that they have defined. ⌋*(RS_SM_00002)*

Component States are used in conjunction with Adaptive Diagnostics to implement means to handle reset requests. For further details see section 7.4.

7.4 Interaction with Adaptive Diagnostics

Adaptive Diagnostics is responsible for diagnosing and configuring and resetting Function Groups. During any diagnostic is executed it is necessary to prevent system from shutting down.

[SWS_SM_0100] Prevent Shutdown due to Diagnostic Session ⌈ State Management shall not shutdown the system during an active diagnostic session. Therefore State Management has to register to Adaptive Diagnostics to receive information about active diagnostic session ⌋*(RS_SM_00100)*

From Adaptive Diagnostics point of view several different reset types have to be carried out to fulfill functionality of Adaptive Diagnostics. Because the interpretation of the reset types (defined in ISO 14229-1)

- hardReset
- keyOffOnReset
- softReset

is done differently by each OEM, parts of the reset functionality have to be delegated by State Management to Adaptive Applications and Adaptive Platform Applications.

Here the Component States comes into scope again: The reset types may be carried out by application when the following reset types are 'translated' by State Managements internal project specific logic to Component States

- hardReset
- softReset

The functionality behind this states is highly Adaptive Application specific. When an Adaptive Application sees no need to support such functionality this states may be skipped immediately. A 'hardReset' could be interpreted e.g. that an Adaptive Application has to reset its related hardware(e.g. a tuner application may reset the tuner hardware by means of the tuner driver). A 'softReset' may be interpreted e.g. that an Adaptive Application has to load its default configuration.

A 'keyOffOnReset' may be translated by State Managements internal logic to stop and start the provided Function Groups.

Please note that this behavior is currently under discussion and therefore subject of change!

[SWS_SM_0101] Diagnostic Reset ⌈ State Management shall implement means to receive reset requests for Function Groups from Adaptive Diagnostics.

State Management shall carry out the project specific actions for the specific reset type ⌋*(RS_SM_00100)*

[SWS_SM_0102] Component States for Reset ⌈ State Management shall provide functionality to enable Components to implement means to execute specific reset types by using Component States. ⌋*(RS_SM_00100, RS_SM_00002)*

The Function Group Machine State has to be handled in a different way when executing reset requests from Adaptive Diagnostics: A 'hardReset' could be interpreted e.g. that an Adaptive Application has to be launched (by requesting e.g. Function Group 'reset' from Execution Management) which carries out the OS or hardware specific reset. A 'softReset' could be interpreted by shutting down all Function Groups and requesting a Machine State 'restart' from Execution Management.

But this functionality is project-specific, too. So therefore the correct mapping has to be done by the OEM code, too.

State Management is the central point in the system, where a reset for the Machine could be requested. So State Management has to keep track of reset causes and has to reset the persistent reset cause when it is newly spawned.

[SWS_SM_0103] Diagnostic Reset Last Cause ⌈ State Management shall provide functionality to persist reset type before Machine reset is carried. ⌋*(RS_SM_00100, RS_SM_00101)*

[SWS_SM_0104] Diagnostic Reset Last Cause Retrieval ⌈ State Management shall read out the last persisted reset cause when State Management is spawned. This reset cause has to be provided via its service interface ⌋*(RS_SM_00100, RS_SM_00101)*

[SWS_SM_0105] Diagnostic Reset Last Cause Reset ⌈ State Management shall reset the last persisted reset cause immediately after State Management has read out the current value. ⌋*(RS_SM_00100, RS_SM_00101)*

7.5 Interaction with Update and Config Management

Update and Config Management is responsible for updating Function Groups, Manifests (execution or machine manifest) or the whole AUTOSAR Adaptive Platform. During any update is executed it is necessary to prevent system from shutting down.

[SWS_SM_0200] Prevent Shutdown due to Update Session ⌈ State Management shall not shutdown the system during an update session is active. therefore State Management has to register to Update and Config Management to receive information about active update session ⌋*(RS_SM_00100)*

To enable Update and Config Management to fulfill its functionality an update and a verify state should be available in the Manifests for each Function Group and for Machine State. Update and Config Management has to request the corresponding Function Group State from State Management.

[SWS_SM_0201] Reset Execution ⌈ State Management shall implement means to issue a Machine reset when Machine State changes from Function Group State 'update' to 'verify'. ⌋*(RS_SM_00100)*

In case of an update of an Adaptive Applications which does not imply an ECU/-machine reset (i.e. soft reset), Execution Management needs to be triggered to reparse the Manifests (execution or machine manifest) of this Adaptive Application prior to restarting it (in order to start it with the right (i.e updated) configuration). Otherwise Execution Management would only reparse the processed Manifests during the next ECU/machine reset (which is too late). For that purpose, an additional interface is needed between Update and Config Management, State Management and Execution Management.

7.6 Interaction with Network Management

To be portable between different ECUs the Adaptive Applications should not have the need to know which networks are needed to fulfill its functionality, because on different ECUs the networks could be configured differently. To control the availability of networks for several Adaptive Applications State Management interacts with Network Management via a service interface. For details see section 9.3.3.

Network Management provides multiple instances of NetworkHandles, where each represents a set of (partial) networks.

The NetworkHandles are defined in the Machine Manifest and are there assigned to a Function Group State.

[SWS_SM_0300] NetworkHandle Configuration ⌈ State Management shall receive information about NetworkHandles and their associated Function Group States from Machine Manifest. ⌋*(RS_SM_00400)*

Whenever (partial) networks are activated or deactivated from outside request and this set of (partial) networks is represented by a NetworkHandle in Machine Manifest Network Management will change the value of the corresponding NetworkHandle. State Management is notified about the change, because it has registered to all availabe NetworkHandle fields. When State Management recognizes a change in a fields value it sets the corresponding Function Group in the Function Group State where the NetworkHandle is configured for in the Machine Manifest.

[SWS_SM_0301] NetworkHandle Registration ⌈ State Management shall register for all NetworkHandles provided by Network Managements which are available from Machine Manifest. ⌋*(RS_SM_00400)*

[SWS_SM_0302] NetworkHandle to FunctionGroupState ⌈ State Management shall set Function Groups to the corresponding Function Group State which is configured in the Machine Manifest for the NetworkHandle when it recognizes a change in NetworkHandle value. ⌋*(RS_SM_00400)*

Vice versa State Managements shall change the value of the NetworkHandle when a Function Group has to change its Function Group State and an association between this Function Group State and the Network handle is available in Machine Manifest. Network Management will recognize this change and will change the state of the (partial) networks accordingly to the NetworkHandle.

[SWS_SM_0303] FunctionGroupState to NetworkHandle ⌈ State Management shall change the value of NetworkHandle when Function Groups changes its Function Group State and a NetworkHandle is associated to this Function Group State in the Machine Manifest. ⌋*(RS_SM_00400)*

It might be needed that a Function Group stays longer in its Function Group State when the causing (partial) network set has been switched off or a (partial) network is longer available than the causing Function Group has been switched to Function Group State 'Off'. This is called 'afterrun'. The corresponding timeout-value has to be configured in Machine Manifest

[SWS_SM_0304] Network Afterrun ⌈ State Management shall support means to support 'afterrun' to switch off related Function Groups or (partial) networks. The timeout value for this 'afterrun' has to be read from e.g. Machine Manifest. ⌋ *(RS_SM_00400)*

7.7 Interaction with Execution Management

Execution Management is used to execute the the Function Group State changes. The decision to change the State of Machine State or the Function Group State of Function Groups might come from inside of State Management based on Operational State (or other project specific requirements) or might be requested at State Management from an external Adaptive Application.

[SWS_SM_0400] Execution Management ⌈ State Management shall use API of Execution Management to change the State of Machine State or Function Group State of Function Groups. ⌋*(RS_SM_00001)*

Execution Management might not be able to carry out the requested Function Group State change due to several reasons (e.g. corrupted binary). Execution Management returns the result of the request.

[SWS_SM_0401] Execution Management Results ⌈ State Management shall evaluate the results of request to Execution Management. Based on this results State Management might decide to do further actions ⌋*(RS_SM_00001)*

[SWS_SM_0402] Function Group State CHange Results ⌈ State Management shall provide Function Group States based on the results of Function Group

State change requests to Execution Management via its service interface ⌋ (RS_SM_00001)

7.8 State Management in a virtualized environment

On an ECU several machines might run in a virtualized environment. Each of the virtual machines might contain an AUTOSAR Adaptive platform. So therefore each of the virtual machines contain State Management. To have coordinated control over the several virtual machines there has to be virtual machine which supervises the whole ECU state.

[SWS_SM_0500] Virtualized State Management ⌈ State Management shall be able to register to a supervising State Management instance to receive information about the whole ECU state. ⌋*(RS_SM_00200, RS_SM_00201)*

[SWS_SM_0501] Virtualized State Management Operational State ⌈ State Management shall implement means to calculate its Operational State based on information from a supervising State Management instance. ⌋*(RS_SM_00200, RS_SM_00201)*

8 API specification

8.1 Type definitions

8.1.1 ComponentState

Name:	ComponentState		
Type:	ara::core::string		
Range:	kInit	'kInit'	Components initial state. Valid till it receives any other state from State Management
	kOff	'kOff'	Component shall prepare for shutdown e.g. persist
	kOn	'kOn'	Component shall operate normally
	kHardReset	'khardReset'	Component shall perform a hard reset (when applicable e.g. reset tuner hardware)
	kSoftReset	'ksoftReset'	Component shall perform a soft reset (when applicable e.g. reset configuration to factory default)
	kFastOff	'kFastOff'	Component shall prepare for shutdown fast e.g. persist partially (e.g. for production diagnosis)
Syntax:	class ComponentState : ara::core::string;		
Header file:	component_client.h		
Description:	Defines the mandatory internal states of a Component (see 7.3).		

Table 8.1: ComponentState

8.1.2 ComponentClientReturnType

Name:	ComponentClientReturnType

Type:	Scoped Enumeration of uint8_t		
Range:	kSuccess	0	component state change successfully completed
	kGeneralError	1	error on interface
	kPending	2	state change accepted for processing, but not accepted for finally being carried out
	kInvalid	3	unknown state
	kAborted	4	confirm of aborted state change due to other state change incoming
	kRejected	5	immediate (in handler function) or delayed (after kPending reporting) information that state was not accepted to be taken. State Management will have to ask again for this state (e.g. with kForced set)
	kUnchanged	6	State was not changed since last request to get new Component State. Only used for polling mode in conjunction with Component-Client::GetComponentState()

Syntax:	`enum class ComponentClientReturnType : uint8_t {`
	`kSuccess = 0,`
	`kGeneralError = 1,`
	`kPending = 2,`
	`kInvalid = 3,`
	`kAborted = 4,`
	`kRejected = 5,`
	`kUnchanged = 6`
	`};`

Header file:	component_client.h
Description:	Defines the error codes for `ComponentClient` operations.

Table 8.2: ComponentClientReturnType

8.1.3 RequestMode

Name:	`RequestMode`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kVoluntary`	`0`	`Component can decline state request`
	`kForced`	`1`	`Component has to carry out state request`

Syntax:	`enum class RequestMode : uint8_t {`
	`kVoluntary = 0,`
	`kForced = 1`
	`};`		

Header file:	component_client.h		
Description:	Defines enforcement options for `ComponentClient` operations.		

Table 8.3: RequestMode

8.1.4 StateUpdateMode

Name:	`StateUpdateMode`		
Type:	`Scoped Enumeration of uint8_t`		
Range:	`kPoll`	`0`	`Component works in polling mode(e.g. safety critical environment)`
	`kEvent`	`1`	`Component works in event mode`

Syntax:	`enum class StateUpdateMode : uint8_t {`
	`kPoll = 0,`
	`kEvent = 1`
	`};`		

Header file:	component_client.h		
Description:	Used to determine if `ComponentClient` operates in polling or event-based mode.		

Table 8.4: StateUpdateMode

8.2 Function definitions

8.2.1 ComponentClient class

The Component State API provides the functionality for a Component to be controlled in a more fine grained way by State Management.

8.2.1.1 ComponentClient::ComponentClient

Service name:	ComponentClient::ComponentClient	
Syntax:	`ComponentClient ara::core::string &s, StateUpdateMode mode);`	
Sync/Async:	Sync	
Parameters (in):	s	Unique name of the Component
	mode	Value of requested operation mode
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	No exceptions thrown	
Description:	Constructor for ComponentClient which opens the State Managements communication channel (e.g. POSIX FIFO) for getting and reporting the Component State. Each Component shall create an instance of this class to get and report its state.	

Table 8.5: ComponentClient::ComponentClient

8.2.1.2 ComponentClient::~ComponentClient

Service name:	ComponentClient::~ComponentClient
Syntax:	`~ComponentClient();`
Sync/Async:	Sync
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	No exceptions thrown
Description:	Destructor for ComponentClient.

Table 8.6: ComponentClient::~ComponentClient

8.2.1.3 ComponentClient::SetStateUpdateHandler

Service name:	ComponentClient::SetStateUpdateHandler	
Syntax:	`ComponentClientReturnType SetStateUpdateHandler( std::function <ComponentClientReturnType( ComponentState &, RequestMode &)> f);`	
Sync/Async:	Sync	
Parameters (in):	f	callback handler for evaluating state changes in event based working mode

Parameters (inout):	None	
Parameters (out):	None	
Return value:	kSuccess	Registration of Component handler was successful
	kInvalid	Registration of Component handler failed due to previous handler registration
	kGeneralError	Registration of Component handler was not successful
Exceptions:	No exceptions thrown	
Description:	Interface for a Component to register its handler to State Management for retival of further Component States in event based working mode.	

Table 8.7: ComponentClient::SetStateUpdateHandler

8.2.1.4 ComponentClient::GetComponentState

Service name:	ComponentClient::GetComponentState	

Syntax:	`ComponentClientReturnType GetComponentState(`
	`ComponentState state,`
	`RequestMode mode);`	

Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	state	Value of the Component State
	mode	used to determine if a Component can decline requested state or not
Parameters (out):	None	
Return value:	kSuccess	Getting new Component State from State Management was successful
	kUnchanged	Getting new Component State from State Management returned no new state
	kGeneralError	Getting new Component State from State Management was failed
Exceptions:	No exceptions thrown	
Description:	Interface for a Component to receive its next internal state from State Management. When mode parameter is set to kForced Component has to enter the provided state(when it is valid). Used only in 'poll' working mode.	

Table 8.8: ComponentClient::GetComponentState

8.2.1.5 ComponentClient::ConfirmComponentState

Service name:	ComponentClient::ConfirmComponentState	

Syntax:	`ConfirmComponentState(`
	`ComponentState &state,`
	`ComponentClientReturnType &status);`	

Sync/Async:	Sync	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	state	Value of the currently reported Component State
	status	Result of a previously Component State request
Return value:	None	
Exceptions:	No exceptions thrown	

Description:	Interface for a Component to report result of Component State request to State Management.

Table 8.9: ComponentClient::ConfirmComponentState

9 Service Interfaces

9.1 Type definitions

This chapter lists all types used in service interfaces of the State Management.

Name	FunctionGroupState	
Kind	TYPE REFERENCE	
Derived from	ara::core::string	
Description	Default FunctionGroup states	
Range	Limit	Description
kOff	’kOff’	FunctionGroup is in Off state
kRunning	’kRunning’	FunctionGroup is in running state
kUpdate	’kUpdate’	FunctionGroup is in Update state
kVerify	’kVerify’	FunctionGroup is in Verify state

Table 9.1: Implementation Data Type - FunctionGroupState

9.2 Provided Interfaces

This chapter lists all provided service interfaces of the State Management.

9.2.1 FunctionGroupState

9.2.1.1 Port

Name	State_{FunctionGroup}		
Kind	Provided Port	***Interface***	FunctionGroupState
Description	Provides handling of FunctionGroupStates		
Variation			

Table 9.2: Port Function Group State - State_{FunctionGroup}

9.2.1.2 Service Interface

Name	FunctionGroupState
NameSpace	ara::sm

Table 9.3: Service Interface State Management - FunctionGroupState

9.2.1.3 Methods

Name	RequestState	
Description	Requests a new Function Group State or Machine State and gathers the ownership for this Function Group	
Parameter	Function Group State or Machine State	
	Description	Function Group State or Machine State to be set.
	Type	FunctionGroupState
	Direction	IN
Application Error Set	FunctionGroupStateErrorSet	

Table 9.4: Service Interface FunctionGroupState - Method RequestState

Name	ReleaseRequest
Description	Releases the current ownership of a Function Group or Machine State.
Parameter	-
Application Error Set	FunctionGroupStateErrorSet

Table 9.5: Service Interface FunctionGroupState - Method ReleaseRequest

9.2.1.4 Fields

Name	FunctionGroupState
Description	Contains the current status of the Function Group {Function Group}
Type	FunctionGroupState
HasGetter	true
HasNotifier	true
HasSetter	false
Init-Value	'kOff'

Table 9.6: Service Interface FunctionGroupState - Field FunctionGroupState

9.2.1.5 Events

Due to the ara field type of Function Group State an event is generated when a Function Group State of a Function Group or the State of Machine State is changed.

9.2.2 DiagnosticReset

9.2.2.1 Port

Name	DiagnosticReset		
Kind	Provided Port	**Interface**	DiagnosticReset
Description	Provides handling of Adaptive Diagnostic reset request		
Variation			

Table 9.7: Port Adaptive Diagnostics - DiagnosticReset

9.2.2.2 Service Interface

Name	Diagnostic Reset
NameSpace	ara::sm

Table 9.8: Service Interface State Management - Diagnostic Reset

9.2.2.3 Methods

Please note that the method to carry out request the diagnostic reset is not yet agreed between Adaptive Diagnostics and State Management .

9.2.3 DiagnosticCommunicationControl

9.2.3.1 Port

Name	DiagnosticCommunicationControl		
Kind	Provided Port	**Interface**	DiagnosticCommunicationControl
Description	Provides handling of Adaptive Diagnostic communication control request		
Variation			

Table 9.9: Port Adaptive Diagnostics - DiagnosticCommunicationControl

9.2.3.2 Service Interface

Name	Diagnostic Communication Control
NameSpace	ara::sm

Table 9.10: Service Interface State Management - Diagnostic Communication Control

9.2.3.3 Methods

Please note that the method to carry out request the communication control is not yet agreed between Adaptive Diagnostics and State Management .

9.3 Required Interfaces

This chapter lists all required service interfaces of the State Management.

9.3.1 Adaptive Diagnostics

Please note that no interface to Adaptive Diagnostics to be informed if a Diagnostic session is ongoing is available in this release.

9.3.2 Update and Config Management

9.3.2.1 Port

Name	PackageManagement		
Kind	Required Port	***Interface***	PackageManagement
Description	provides information about current status of update session		
Variation			

Table 9.11: Port Update and Config Management - PackageManagement

9.3.2.2 Service Interface

Name	PackageManagement
NameSpace	ara::ucm::pkgmgr

Table 9.12: Service Interface Update and Config Management - PackageManagement

9.3.2.3 Fields

Name	CurrentStatus
Description	Set to kIdle by Update and Config Management when not active. Any other value is usedto prevent shutdown of the system during active update session
Type	PackageManagerStatusType
HasGetter	true
HasNotifier	true
HasSetter	false
Init-Value	false

Table 9.13: Service Interface Update and Config Management - Field CurrentStatus

9.3.3 Network Management

9.3.3.1 Port

Name	NetworHandle_{NmNode}		
Kind	Required Port	***Interface***	NetworkHandle
Description	contains information about active network handle		
Variation			

Table 9.14: Port Network Management - NetworkHandle

9.3.3.2 Service Interface

Name	NetworkHandle
NameSpace	ara::nm

Table 9.15: Service Interface Network Management - NetworkHandle

9.3.3.3 Fields

Please note that this field is not yet finally discussed and therefore subject of change.

Name	NetworkHandle
Description	Set to true by Network Management when (partial) networks are active or by State Management to activate (partial) networks
Type	BooleanType
HasGetter	true
HasNotifier	true
HasSetter	true
Init-Value	false

Table 9.16: Service Interface Network Management - Field NetworkHandle

9.4 Application Errors

This chapter lists all application errors of State Management.

9.4.1 Application Error Domain

9.4.1.1 FunctionGroupStateErrors

Name	*Code*	*Description*
kSuccess	0	FunctionGroup State change request was executed successfully
kInvalid	1	FunctionGroup State change request was invalid e,g, unknown state
kFailed	2	FunctionGroup State change request failed due to other reason
kDelay	3	FunctionGroup State change request was delayed due to ownership

Table 9.17: Application Errors of FunctionGroupState

9.4.2 Application Error Set

9.4.2.1 FunctionGroupStateErrorSet

Error Set Name	FunctionGroupStateError
Description	The potential errors values returned using State Managements Function Group State change service interface
Reference	kInvalid, kFailed, kDelay

Table 9.18: Application Error Set of FunctionGroupState

A Used Interfunctional Cluster Interfaces

B Not applicable requirements