Document Title	Demonstrator Design of Functional Cluster Update And Configuration Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	894

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2020-11-30	R20-11	AUTOSAR Release Management	• Improved state machine implementation • Partial implementation of vehicle package manager • Migrated shared code of UCM and VPM into library • Reworked document structure
2019-11-28	R19-11	AUTOSAR Release Management	• Implemented package processing • Implemented ApApplicationError service interface • Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Implemented state management • Implemented software cluster management • Extended package manifest and layout
2018-11-02	18-10	AUTOSAR Release Management	• Main function and service offering provided • Data transfer sequence supported • Std types replaced with ara::core types

2018-05-02	18-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document describes the design (approach and decisions) of the Functional Cluster Update and Configuration Management for AUTOSAR's Adaptive Platform Demonstrator.

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply other implementations as the standard is defined by the specifications. Nevertheless, the Demonstrator may supplement and ease the understanding of the specifications.

This implementation of Update and Configuration Management software provides a complete and well-documented reference implementation that is meant to be used in prototype projects as well as a basis for production-grade implementations. The main goal here was to provide proof of concepts for Update and Configuration Management features specified in the Update and Configuration Management specification of AUTOSAR Adaptive. Optimization for memory consumption and speed have been considered but they haven't played a major role in overall software design and implementation. More than that, no static or dynamic code analysis has been performed, therefore any series production project deriving form this implementation, will have to further fulfill the safety constraints described in the industry standards.

1.1 Known limitations

Limitations of the Demonstrator implementation of the Functional Cluster.

Only the following requirements are implemented completely. Some other requirements are implemented partially.

Validation set	Spec item
	SWS_UCM_00099
	SWS_UCM_00103
	SWS_UCM_00136
ucm_ UCM identification	SWS_UCM_00009
ucm_Activation and Rollback	SWS_UCM_00005
ucm_Activation and Rollback	SWS_UCM_00020
ucm_Activation and Rollback	SWS_UCM_00022
ucm_Activation and Rollback	SWS_UCM_00025
ucm_Activation and Rollback	SWS_UCM_00026
ucm_Applications Persisted Data	SWS_UCM_00184
ucm_Content of Software Package	SWS_UCM_00112
ucm_Logging and history	SWS_UCM_00115
ucm_Logging and history	SWS_UCM_00160
ucm_Processing Software Packages	SWS_UCM_00001
ucm_Processing Software Packages	SWS_UCM_00003

△

ucm_Processing Software Packages	SWS_UCM_00018
ucm_Processing Software Packages	SWS_UCM_00024
ucm_Processing Software Packages	SWS_UCM_00029
ucm_Processing Software Packages	SWS_UCM_00104
ucm_Processing Software Packages	SWS_UCM_00137
ucm_Processing Software Packages	SWS_UCM_00161
ucm_Software Package	SWS_UCM_00122
ucm_Status Reporting	SWS_UCM_00017
ucm_Status Reporting	SWS_UCM_00019
ucm_Status Reporting	SWS_UCM_00080
ucm_Status Reporting	SWS_UCM_00081
ucm_Status Reporting	SWS_UCM_00083
ucm_Status Reporting	SWS_UCM_00084
ucm_Status Reporting	SWS_UCM_00086
ucm_Status Reporting	SWS_UCM_00126
ucm_Status Reporting	SWS_UCM_00127
ucm_Status Reporting	SWS_UCM_00146
ucm_Status Reporting	SWS_UCM_00147
ucm_Status Reporting	SWS_UCM_00149
ucm_Status Reporting	SWS_UCM_00150
ucm_Status Reporting	SWS_UCM_00151
ucm_Status Reporting	SWS_UCM_00152
ucm_Status Reporting	SWS_UCM_00154
ucm_Status Reporting	SWS_UCM_00162
ucm_Status Reporting	SWS_UCM_00163
ucm_Status Reporting	SWS_UCM_00164
ucm_Transferring Software Packages	SWS_UCM_00007
ucm_Transferring Software Packages	SWS_UCM_00008
ucm_Transferring Software Packages	SWS_UCM_00010
ucm_Transferring Software Packages	SWS_UCM_00021
ucm_Transferring Software Packages	SWS_UCM_00075
ucm_Transferring Software Packages	SWS_UCM_00087
ucm_Transferring Software Packages	SWS_UCM_00088
ucm_Transferring Software Packages	SWS_UCM_00140
ucm_Transferring Software Packages	SWS_UCM_00145
ucm_Transferring Software Packages	SWS_UCM_00148
ucm_Version Reporting	SWS_UCM_00004
ucm_Version Reporting	SWS_UCM_00030
ucm_Version Reporting	SWS_UCM_00185

2 Overview on architecture

This chapter gives an overview on the UCM architecture. It covers **Interfaces to other functional clusters** (2.1), information on the **Design approach** (2.2), **Used OSS components** (2.3) and a **Class Overview** (3.4).

2.1 Interfaces to other functional clusters

UCM has multiple interfaces to other functional clusters on the Adaptive Platform as depicted in the figure below. The following subsections describe the implemented as well as the planned interfaces.

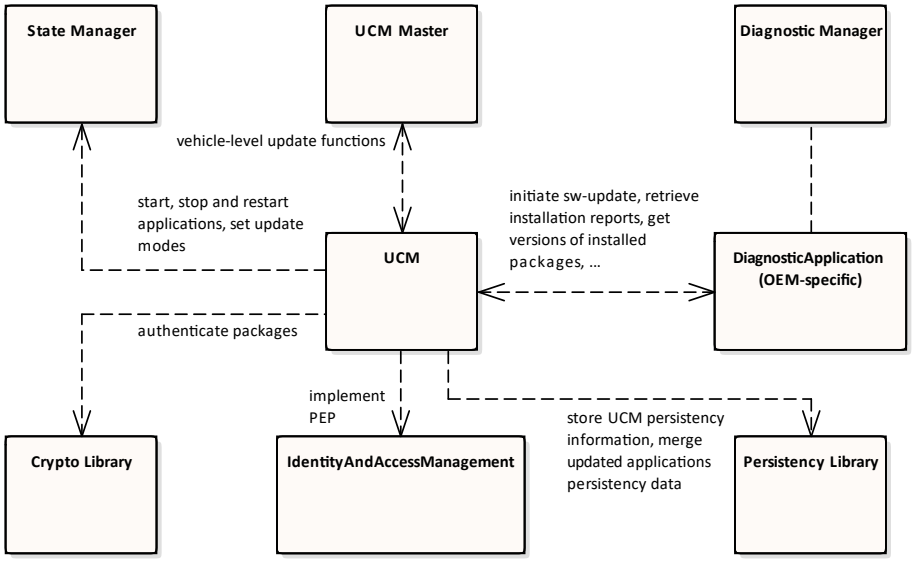

Figure 2.1: Interfaces to other functional clusters

2.1.1 UCM Master

UCM service interface can be used by UCM Master to implement vehicle-level update functions. UCM Master is further referred to as as Vehicle Package Manager (VPM) in this document.

2.1.2 State Management

For full operation, UCM has to be able to set machine modes, start, stop or restart Adaptive Applications or Functional Clusters. This is done using interfaces towards State Management. As some of the needed interfaces do not yet exists, calls to State Management are currently stubbed.

2.1.3 Persistency

UCM currently has no implemented interface towards Persistency. Later releases may contain functionality to merge existing persistency data of an Adaptive Application with persistency data supplied with a Software Package.

2.1.4 Diagnostics

In order to support Software Clusters having an own diagnostics target address UCM needs to have interfaces to functional cluster Diagnostics. These interfaces shall be used to inform Diagnostics about newly installed or removed Software Clusters so that Diagnostics can properly handle requests targeted to them.

As UCM currently only implements Software Clusters without a dedicated diagnostics target address these interfaces are neither defined nor implemented.

2.1.5 Identity and Access Management

UCM currently has no implemented interface towards Identity and Access Management. An implementation of a Policy Enforcement Point (PEP) is planned for later releases.

2.1.6 Crypto library

UCM needs to verify authentication of received packages using the `ara::crypto` library.

As the crypto library is not yet available this is planned for later releases.

2.2 Design approach

2.2.1 Components

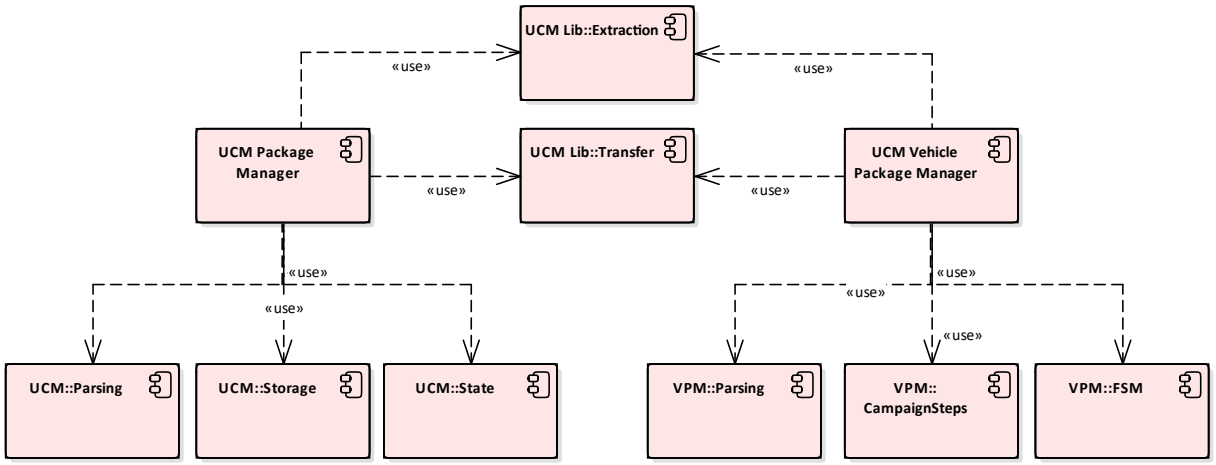

Figure 2.2: Update And Configuration Management software - high level component diagram

- **Package Manager** software component is the central component of UCM implementation. It has the following components.
  - **Parsing** handles the packages and UCM's platform metadata
  - **Storage** is responsible for writing packages to the Non-Volatile Memory (NVM)
  - **StateMgmt** handles internal state dependent behaviour
- **Vehicle Package Manager** software component is the central component of UCM master implementation. It has the following components.
  - **Parsing** handles the vehicle package metadata
  - **CampaignSteps** handles the campaign steps created from the vehicle package input
  - **FSM** implements the state machine of VPM
- **UCM Lib** contains shared functionality used by both UCM and VPM
  - **Transfer** implements different means to receive an update
  - **Extraction** implements different extraction methods

The Update And Configuration Management cluster processes software clusters containing a set of applications. It offers services for installing, uninstalling and updating software clusters and their content.

2.3 Used OSS components

This sections lists all OSS components used in the implementation of the Functional Cluster, their architectural context and the rationale for their use.

OSS	Version	Explanation	Binding	License	License Version
-Boost-	-1.68.00-	-Filesystem access-	-header only, static linked-	-Boost Software License-	-1.0-
-Google Test-	-1.8.0-	-Unit testing-	-Not relevant-	-3-Clause BSD-	-NA-
-POCO-	-1.9.0-	-Archive extraction-	-Dynamic linking-	-Boost Software License-	-1.0-

2.4 Safety and security aspects of the implementation

This section covers security and safety issues of the current implementation.

2.4.1 Security

As `ara::crypto` is not yet available UCM does not implement or employ signature and authorization checks offered by it. This is planned for later releases.

There is currently no enforced authorization check of requests to UCM's service interface. In particular, the transfer id assigned to the client on TransferStart and used in subsequent calls to other methods of the service interface shall not be regarded as a security feature. For future releases, an implementation of a PEP shall adress these authorization issues.

2.4.2 Safety

The implementation is currently not safe against interruption of package processing and may not be able to recover from power loss or similar in all operation states. In particular, if power loss occurs during processing, activation or rollback phase, UCM will be in an inconsistent state as no recovery mechanisms are implemented.

3 UCM design aspects

This chapter details the design approach of the UCM implementation.

3.1 State management

State management is implemented with specialized classes derived from generic class PackageManagerState. The default implementations return error codes. States which support the callable functions override these in their internal implementation. The following figure shows which methods are overridden.

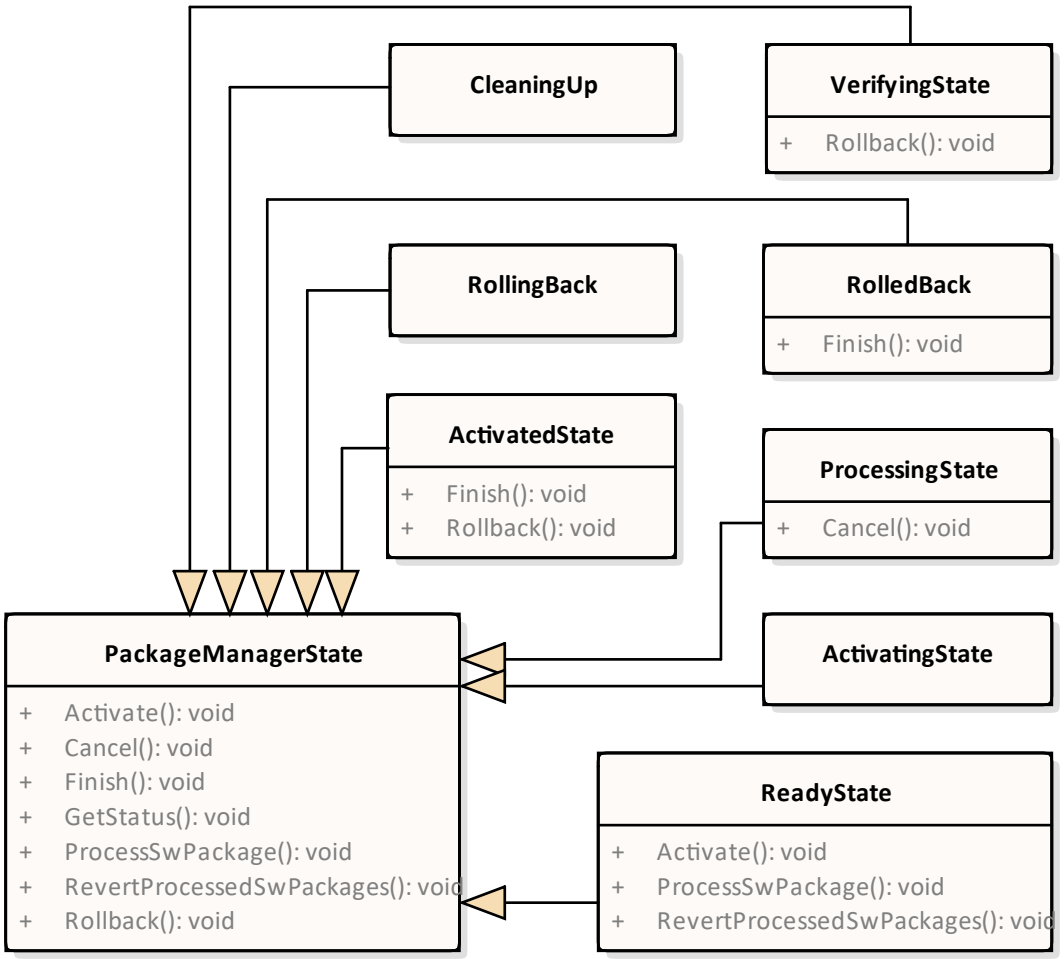

Figure 3.1: State management classes

The following table describes the implemented state transitions. The event column lists the possible external events *Activate*, *Cancel*, *Finish*, *ProcessSwPackage*, *Revert ProcessedSwPackages* and *Rollback* with a prefix '<' to indicate incoming direction. Internal events are prefixed with '$' resp. '!' for error events. The columns current state and next state list the state transition. The action column shows which action is performed in the state. The prefix '>' shows outgoing messages to the client.

Most client requests are answered after a state transition, e.g. the call to *Activate* is received in state *Ready* but answered in *Activating*.

Other external events, e.g. *TransferStart* have no influence on the state changing and are therefore not listed here to enhance clarity.

Event	Current state	Next state	Action
<Activate	Activated	Activated	>ApErr: OperationNotPermitted
<Cancel	Activated	Activated	>ApErr: OperationNotPermitted
<Finish	Activated	Cleaning-Up	State change
<ProcessSwPackage	Activated	Activated	>ApErr: OperationNotPermitted
<RevertProcessedSwPackages	Activated	Activated	>ApErr: OperationNotPermitted
<Rollback	Activated	Rolling-back	State change
$EnterState from Ready	Activating	Activating	Check dependencies
$Success	Activating	Verifying	>Activate
!PreActivationFailed	Activating	Ready	>ApErr (Activate): PreActivationFailed
!ErrorNoValidProcessing	Activating	Ready	>ApErr (Activate): ErrorNoValidProcessing
!MissingDependencies	Activating	Ready	>ApErr (Activate): MissingDependencies
<Activate	Activating	Activating	>ApErr: OperationNotPermitted
<Cancel	Activating	Activating	>ApErr: OperationNotPermitted
<Finish	Activating	Activating	>ApErr: OperationNotPermitted
<ProcessSwPackage	Activating	Activating	>ApErr: OperationNotPermitted
<RevertProcessedSwPackages	Activating	Activating	>ApErr: OperationNotPermitted
<Rollback	Activating	Activating	>ApErr: OperationNotPermitted
$EnterState from Activated (Finish)	Cleaning-Up	Cleaning-Up	Finalize the activation: (CommitChanges() for each processed Action)
$EnterState from Ready (RevertProcessedSw Packages)	Cleaning-Up	Cleaning-Up	Finalize the revert: (RevertChanges() for each processed Action)
$EnterState from Rolled-back (Finish)	Cleaning-Up	Cleaning-Up	Finalize the rollback: (RevertChanges() for each processed Action)
$Success Finish	Cleaning-Up	Idle	>Finish
$Success Revert ProcessedSwPackages	Cleaning-Up	Idle	>RevertProcessedSwPackages
!NotAbleToRevertPackages (for Revert ProcessedSwPackages)	Cleaning-Up	Ready	>ApErr (RevertProcessedSwPackages): NotAbleToRevertPackages
!NothingToRevert (for RevertProcessedSw Packages)	Cleaning-Up	Ready	>ApErr (RevertProcessedSwPackages): NothingToRevert
<Activate	Cleaning-Up	Cleaning-Up	>ApErr: OperationNotPermitted
<Cancel	Cleaning-Up	Cleaning-Up	>ApErr: OperationNotPermitted

△

<Finish	Cleaning-Up	Cleaning-Up	>ApErr: OperationNot Permitted
<ProcessSwPackage	Cleaning-Up	Cleaning-Up	>ApErr: OperationNot Permitted
<RevertProcessedSw Packages	Cleaning-Up	Cleaning-Up	>ApErr: OperationNot Permitted
<Rollback	Cleaning-Up	Cleaning-Up	>ApErr: OperationNot Permitted
<Activate	Idle	Idle	>ApErr: OperationNot Permitted
<Cancel	Idle	Idle	>ApErr: OperationNot Permitted
<Finish	Idle	Idle	>ApErr: OperationNot Permitted
<ProcessSwPackage	Idle	Processing	State change
<RevertProcessedSw Packages	Idle	Idle	>ApErr: OperationNot Permitted
<Rollback	Idle	Idle	>ApErr: OperationNot Permitted
$EnterState from Ready	Processing	Processing	Create cancelable processing thread and start processing
$Success Cancel	Processing	Processing	>Cancel
$Success ProcessSw Package	Processing	Ready	>ProcessSwPackage
!CancelFailed (for Cancel)	Processing	Processing	>ApErr (Cancel): CancelFailed
!InsufficientMemory	Processing	Ready/Idle	>ApErr (ProcessSw Package): InsufficientMemory
!InvalidManifest	Processing	Ready/Idle	>ApErr (ProcessSw Package): Invalid Manifest
!InvalidTransferId (for Cancel)	Processing	Processing	>ApErr (Cancel): InvalidTransferId
!InvalidTransferId (for ProcessSwPackage)	Processing	Ready/Idle	>ApErr (ProcessSw Package): Invalid TransferId
!ProcessedSwPackage Inconsistent	Processing	Ready/Idle	>ApErr (ProcessSw Package): Processed SwPackageInconsistent
!ProcessSwPackage Cancelled	Processing	Ready/Idle	>ApErr (ProcessSw Package): ProcessSw PackageCancelled
<Activate	Processing	Processing	>ApErr: OperationNot Permitted
<Cancel	Processing	Processing	Set cancel flag
<Finish	Processing	Processing	>ApErr: OperationNot Permitted
<ProcessSwPackage	Processing	Processing	>ApErr (ProcessSw Package): Service Busy
<RevertProcessedSw Packages	Processing	Processing	>ApErr: OperationNot Permitted
<Rollback	Processing	Processing	>ApErr: OperationNot Permitted

▽

△

<Activate	Ready	Activating	State change
<Finish	Ready	Ready	>ApErr: OperationNot Permitted
<ProcessSwPackage	Ready	Processing	State change
<RevertProcessedSw Packages	Ready	Cleaning-Up	State change
<Rollback	Ready	Ready	>ApErr: OperationNot Permitted
<Activate	Rolled-back	Rolled-back	>ApErr: OperationNot Permitted
<Cancel	Rolled-back	Rolled-back	>ApErr: OperationNot Permitted
<Finish	Rolled-back	Cleaning-Up	State change
<ProcessSwPackage	Rolled-back	Rolled-back	>ApErr: OperationNot Permitted
<RevertProcessedSw Packages	Rolled-back	Rolled-back	>ApErr: OperationNot Permitted
<Rollback	Rolled-back	Rolled-back	>ApErr: OperationNot Permitted
$EnterState from Verifying or Activated	Rolling-back	Rolling-back	Do the rollback (previous state of applications restored)
$Success	Rolling-back	Rolled-back	>Rollback
!NotAbleToRollback from Activated	Rolling-back	Activated	>ApErr (Rollback): NotAbleToRollback
!NotAbleToRollback from Verifying	Rolling-back	Verifying	>ApErr (Rollback): NotAbleToRollback
!NothingToRollback from Activated	Rolling-back	Activated	>ApErr (Rollback): NothingToRollback
!NothingToRollback from Verifying	Rolling-back	Verifying	>ApErr (Rollback): NothingToRollback
<Activate	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
<Cancel	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
<Finish	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
<ProcessSwPackage	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
<RevertProcessedSw Packages	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
<Rollback	Rolling-back	Rolling-back	>ApErr: OperationNot Permitted
$EnterState from Activating	Verifying	Verifying	Start thread to monitor application / platform restart
$Success	Verifying	Activated	State change
!Fail	Verifying	Rolling-back	State change
<Activate	Verifying	Verifying	>ApErr: OperationNot Permitted
<Cancel	Verifying	Verifying	>ApErr: OperationNot Permitted
<Finish	Verifying	Verifying	>ApErr: OperationNot Permitted

▽

△

<ProcessSwPackage	Verifying	Verifying	>ApErr: OperationNotPermitted
<RevertProcessedSwPackages	Verifying	Verifying	>ApErr: OperationNotPermitted
<Rollback	Verifying	Rolling-back	Cancel monitor thread and do state change

3.2 Data flow

UCM basically has two data stores used for operation. First there is a cache used to temporarily store transferred packages. The cache is used for all clients, i.e. they can transfer packages in parallel as long as storage and processing resources for the transfer are available.

This cache is then used for actual processing of the packages, i.e. making modifications to the platform's NVM storage. The NVM storage is also used to store package processing logs which enable investigation of error cases. The report process reads from both NVM and also the cache to answer client requests on available software. It also makes history data on package processing available to the client.

UCM's API is intended to support an "A/B" NVM schema to update on inactive storage and then switch that storage to become the active one. This could be implemented e.g. using two partitions in a dual bank setup, a filesystem supporting snapshots and subvolumes like btrfs, a versioning system like OSTree or a directory hierarchy.

A concrete implementation is to a certain degree hardware dependent as it has to interact with the bootloader, i.e. to choose active partitions and configure watchdogs.

The demonstrator uses a custom transaction system based on versioned directories with an atomically switched index file. It is described in **Base technology for A/B switching** (4.1).

The following figure shows the data flow for UCM.

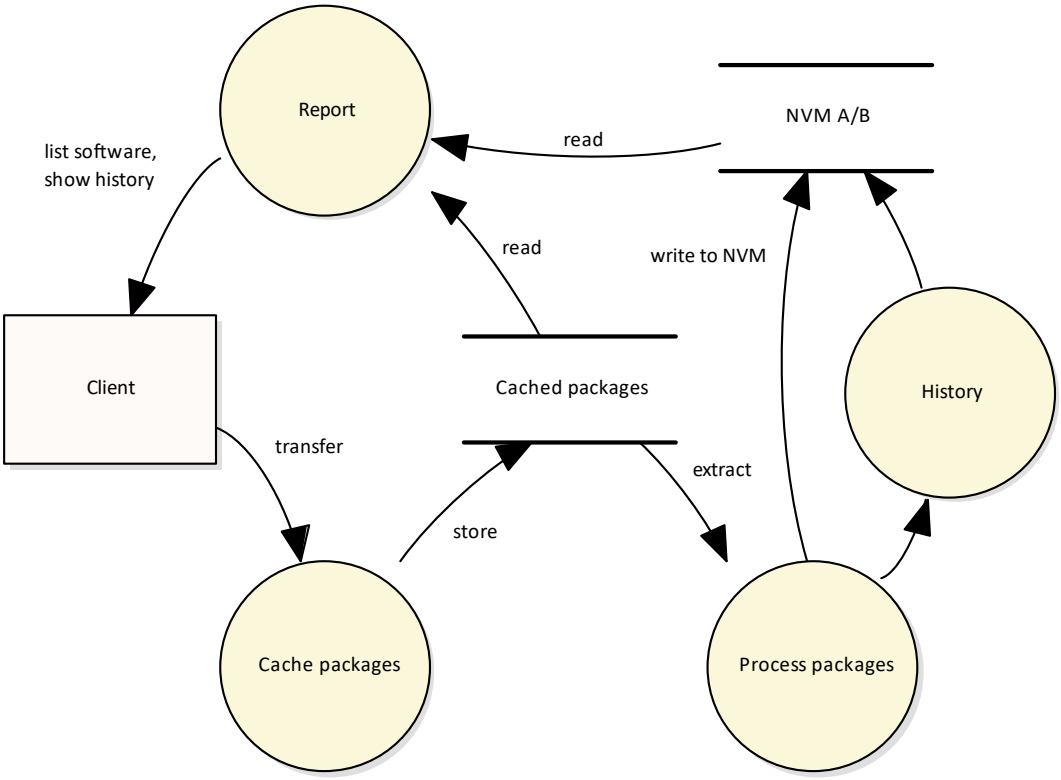

Figure 3.2: Data flow of Package Manager

3.3 Control flow

Due to safety requirements in case of interrupted updates UCM package manager's interface towards Diagnostic Service Applications (DSA) is very state dependent.

Currently only version requests and logging can be called independent of the state. Therefore the internal implemenation also follows a specific path. The following picture shows a control flow diagram of the Package Manager.

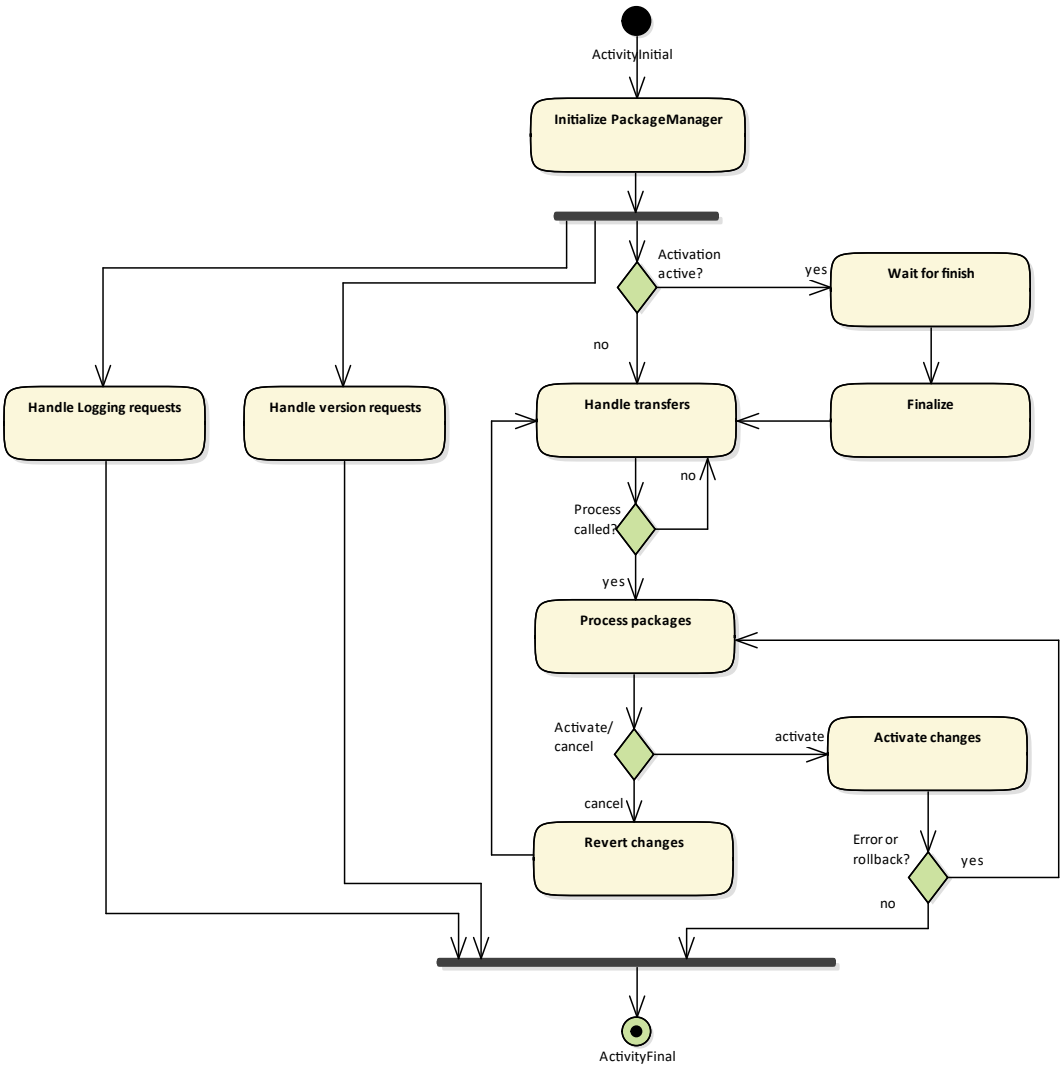

Figure 3.3: Control flow of Package Manager

3.4 Class Overview

3.4.1 Package Manager

Package Manager is the main deliverable provided by Update And Configuration Management functional cluster. It implements most of the features described in the Update And Configuration Management specification.

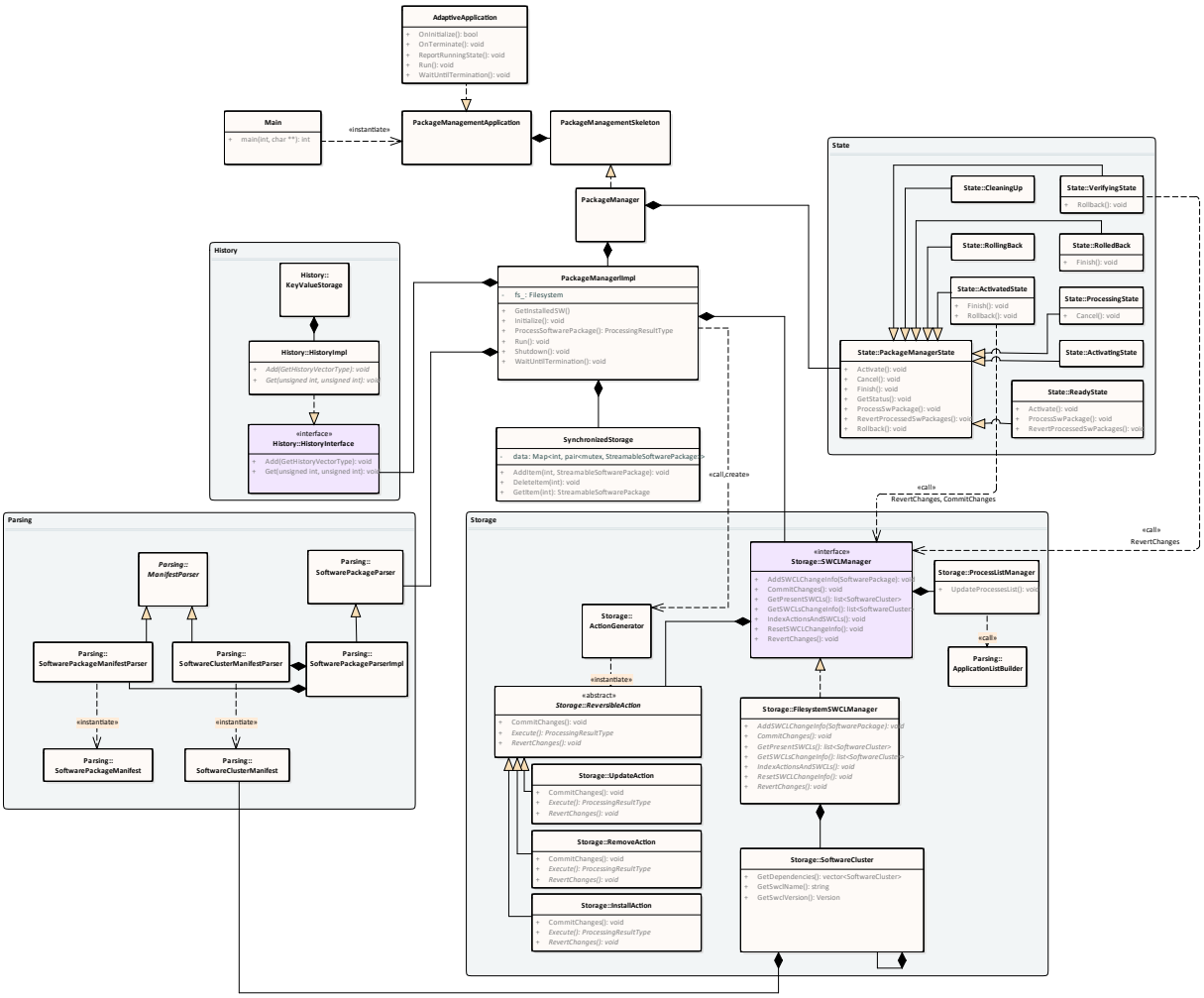

Figure 3.4: Package Manager - class diagram

The Package Manager instantiates a PackageManagementApplication object that handles the operation of the service interface and signals (e.g. SIGTERM).

The main service provided by UCM's Package Manager is ProcessSwPackage.

The function extracts a software package to a temporary location. It then parses the software package manifest to check the operation type requested.

3.4.2 Extraction component

As the package archive format is not yet fixed or may be left as implementation specific the extraction is abstracted to an interface that can be implemented by different providers. The demonstrator currently implements the LibPocoZipExtractor.

The following figure shows the component overview.

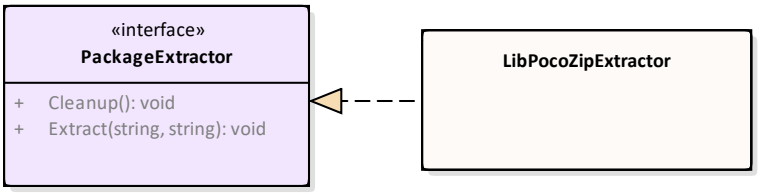

Figure 3.5: Extract component

3.4.3 Parsing component

The parsing component has interfaces to the Package Manager which enable the latter one to get information on the installed components. The SoftwareClusterListBuilder indexes the content of the platform and makes it available via the version report interface. For implementing the A/B functionality two instances are run, one on the active and the other on the inactive partition.

The following figure shows the component overview.

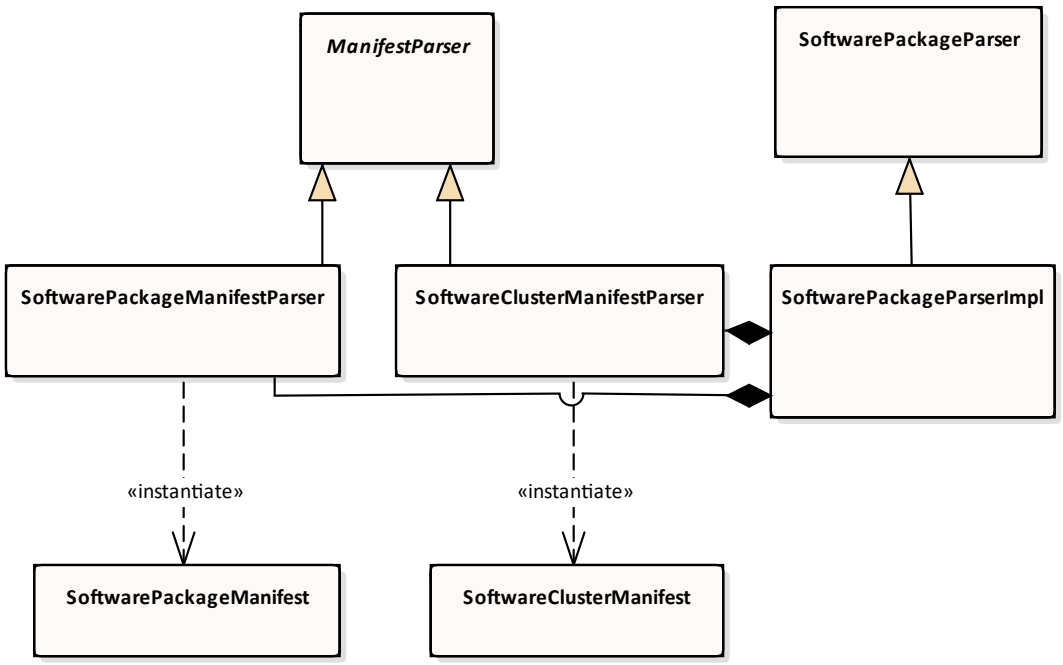

Figure 3.6: Parsing component

3.4.4 Storage components

As a software package can contain an **InstallAction** (3.4.4.1), a **RemoveAction** (3.4.4.2), or an **UpdateAction** (3.4.4.3) request, different actions are created by the action generator depending on the input SoftwarePackage passed to the PackageManager's ProcessSwPackage() method.

The ReversibleAction class defines three virtual methods to be implemented by subclasses:

- An Execute() method which is called when the ProcessSwPackage() is called on that package.
- A RevertChanges() method which reverts all changes done in the Execute method.
- A CommitChanges() method which finalizes the changes done in the Execute method.

The methods RevertChanges() and CommitChanges() are called when entering the CleaningUp state. If entering the state via a RevertProcessedSwPackages() call from the client RevertChanges() is called. If entering the state due to a Finish() call the executed method depends on whether a rollback was done.

	RevertChanges()	CommitChanges()
Finish()		X
Finish() after Rollback()	X	
RevertProcessedSwPackages()	X	

The storage component implements the class SWCLManager which indexes the platforms NVM for software clusters directories and parses their manifest files. It provides function AddSWCLChangeInfo() which adds the changes done by a specific software package. This information is accessed by GetSwClusterChangeInfo(). After Finish() the change information is reset by calling ResetSWCLChangeInfo() and the SWCLManager reindexes the present SWCLs to answer client requests to GetSw ClusterInfo().

The following figure shows the component overview.

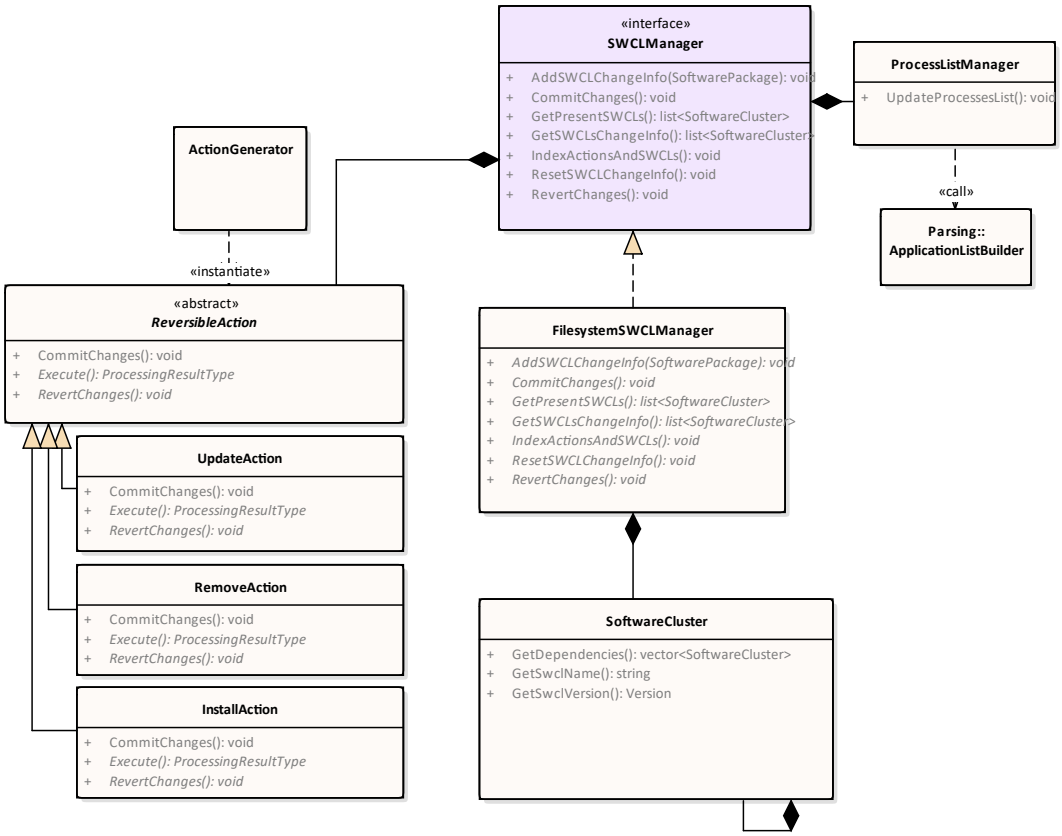

Figure 3.7: Storage component

3.4.4.1 InstallAction

In case of an InstallAction, the SoftwarePackage object created by the parsing component is processed by the action. It contains references to the SoftwareCluster's artifacts that will be written to the filesystem. For the Execute method it moves the extracted software cluster directory to the final location into a subdirectory named by its version number, see file_system_layout for details. A reference is added in the SWCLManager by calling AddSWCL with the state set to kAdded. For the RevertChanges method it removes the newly created directory and the reference in the SWCLManager. On CommitChanges no action is taken as the software cluster is kept and its state changes to kPresent in the SWCLManager.

3.4.4.2 RemoveAction

In case of a RemoveAction, the Execute method sets the software cluster as kRemoved in the SWCLManager. The RevertChanges sets its state back to kPresent. On Commit Changes the software cluster directory is removed. The software cluster is removed from the SWCLManager.

3.4.4.3 UpdateAction

An update extracts the software clusters new version to the file system and sets the associated Software Cluster object in the SWCLManager to state kUpdated on Execute call. On RevertChanges the new version is removed again from the filesystem. On CommitChanges the old version is removed from the filesystem.

The following table lists the filesystem actions and the state to set the software cluster to for each method.

	InstallAction	RemoveAction	UpdateAction
Execute	move / kAdded	- / kRemoved	move (new) / kUpdated
RevertChanges	delete / -	- / kPresent	delete (new) / kPresent
CommitChanges	- / kPresent	delete / -	delete (old) / kPresent

Actions are owned by the FilesystemSWCLManager and implicitly form a transaction supported by the state machine flow.

3.4.4.4 Activate and Rollback

After processing is finished activation can be performed. In this step the package manager performs dependency checks of the installed software for all software clusters with state kAdded, kUpdated or kPresent. If the check is successful it creates a new processes list including the installed and updated processes whose software cluster are not in state kRemoved as well as the platform-level applications. After the file is generated it is switched with the current processes list which is still kept for a possible rollback.

In case of rollback the backup process list becomes active again and is send to EM.

3.4.4.5 Finish

If Finish() has been called after Rollback() it removes everything that was added during the prior processing, i.e. updated and newly installed software clusters by calling the RevertChanges function for each ReversibleAction and deletes the newly created processes list.

If Finish() has been called after Activate() without a rollback it removes uninstalled software clusters and old versions of updated software clusters by calling CommitChanges for each ReversibleAction. The old version of the processes list is also deleted.

3.5 Usage of the service interface

UCM's Package Manager offers a service interface via ara::com. The sample application is implemented against this interface and executes a basic update sequence. The sequence is documented in SWS_UpdateAndConfigManagement_888, in particular refer to chapter 10 "Sequence diagrams".

3.6 Interfaces to the packaged application

UCM specification for this release does not contain interfaces towards a packaged application which the latter could use for example to execute OnUpdate calls, e.g. to migrate its own persistency data to a new structure. Therefore also the demonstrator implementation does not contain interfaces towards packaged applications at this point.

4 UCM implementation details

4.1 Base technology for A/B switching

UCM implements A/B switching using version-tagged software cluster directories inside the platform's filesystem and a common file which lists the active software clusters processes in **Processes list** (4.1.2).

UCM does currently not modify other Functional Clusters, nor system components like kernel and base libraries.

4.1.1 Filesystem layout

UCM installs each software cluster by shortname in a different directory. Subdirectories are used for different versions so they can be installed side by side to enable rollback. If `Finish()` is called the old/removed version is removed from the filesystem or in case of previous rollback the updated/installed version is removed.

To illustrate this a simple example is used. Consider a system which has these software clusters installed:

Software cluster shortname	Version
swcl_parkassist	1.0.0.3
swcl_camera	1.0.4.0

In the processing phase swcl_parkassist is updated, swcl_camera is removed and swcl_radar is added because the parkassist now uses a radar instead of a camera:

Software cluster shortname	Version
swcl_parkassist	1.0.0.4
swcl_radar	1.0.0.0

The following image shows the structure on the filesystem after activation has been called.

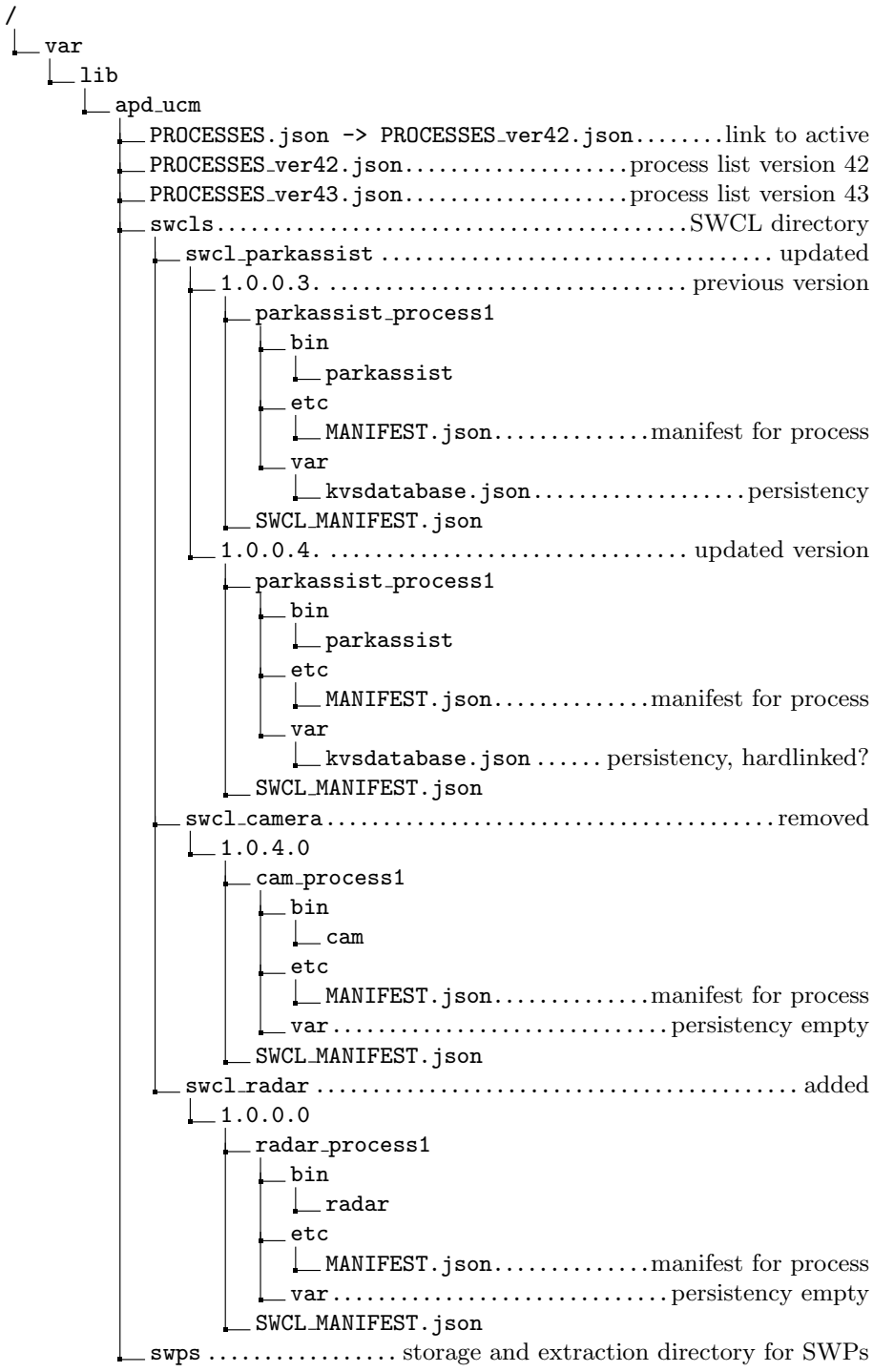

Figure 4.1: Directory structure for software clusters

The two versions of swcl_parkassist are installed side by side, swcl_camera is still on the file system and swcl_radar has been extracted to the file system. A new process list was created in `PROCESSES_ver43.json` and the link is switched to the new version. Execution management could now be notified to use the updated processes list to start newly installed processes and/or restart updated ones.

If the system does a rollback to the old state `PROCESSES.json` is linked back to `PROCESSES_ver42.json`.

Upon finish the obsolete software cluster versions and the obsolete processes list is removed again.

4.1.2 Processes list

The processes list contains all activated platform processes. In the example use case it would look like this before the processing:

```
[
    {
        "key": "processes",
        "value": {
            "string[]": [
                "/opt/radar/",
                "/opt/fusion/",
                "/opt/ucm/",
                "/var/lib/apd_ucm/swcl_parkassist/1.0.0.3/parkassist_process1",
                "/var/lib/apd_ucm/swcl_camera/1.0.4.0/cam_process1"
            ]
        },
        "checksum": 870226170
    }
]
```

The new file is created after Activate() is called. It contains updated paths to the new version of swcl_parkassist, removed the entry for swcl_camera's processes and a new entry for the newly installed swcl_radar.

```
[
    {
        "key": "processes",
        "value": {
            "string[]": [
                "/opt/radar/",
                "/opt/fusion/",
                "/opt/ucm/",
                "/var/lib/apd_ucm/swcl_parkassist/1.0.0.4/parkassist_process1",
                "/var/lib/apd_ucm/swcl_radar/1.0.0.0/radar_process1"
            ]
        },
        "checksum": 8357487
    }
```

15]

4.2 Package format

The demonstrator implementation of Update And Configuration Management uses a vendor-specific package format consisting of a Zip archive with *defined order of elements* to enable processing in streaming mode. The following image shows its internal structure and ordering.

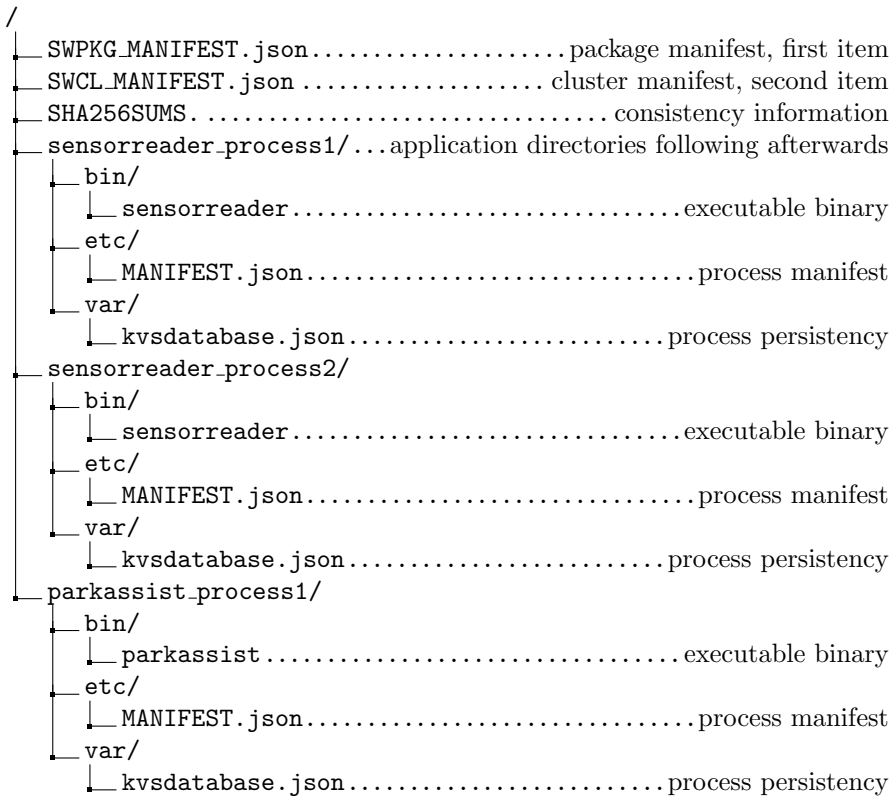

Figure 4.2: UCM Software Package Content

Each Software Package addresses one Software Cluster. For every process inside the software package it contains subdirectories with the process executables and their configuration or resource files. Although processes might share the same binary as in the example, there is currently no use of references for this in the packaging nor when writing them to the filesystem (e.g. using hard-/ or softlinks).

Authentication and integrity protection of data is currently not supported for the demonstrator implementation due to missing Crypto library.

The contents of the manifest files are explained in the following subsections.

4.2.1 Software Package manifest

The software package manifest contains meta information about the software cluster contained in a software package. The current implementation uses a JSON representation of the ARXML definition in the metamodel. Some attributes like compressed SoftwarePackageSize and uncompressedSoftwarePackageSize are not contained in the manifest as they can be determined from the size of the archive and its metadata.

The following listing shows the Software Package manifest (`SWPKG_MANIFEST.json`) for an installation package. Consistency information is embedded into the manifest in the key `files` using CRC32 hashes. The generation is supported by tooling #package_generator.

```
{
    "actionType": "Install",
    "activationAction": "restartApplication",
    "deltaPackageApplicableVersion": "0.0.9-1547023846",
    "minUCMSupportedVersion": "1.0.0-0",
    "maxUCMSupportedVersion": "1.0.0-0",
    "packagerID": "1347111236",
    "postVerificationReboot": "false",
    "preActivate": "mode1",
    "preActivationReboot": "false",
    "shortName": "SWP_TEST1_INS",
    "uncompressedSoftwareClusterSize": 12395621,
    "uuid": "079af322-1570-11e9-8e05-e79fcc9bd29c",
    "verify": "mode2",
    "files": {
        "swcl_sample": {
            "bin": {
                "swcl_sample": "1f43ff3f"
            },
            "etc": {
                "MANIFEST.json": "7a0cf69c"
            }
        },
        "SWCL_MANIFEST.json": "010ad84f"
    }
}
```

The Software Cluster manifest for this package (`SWCL_MANIFEST.json`) contains this information.

```
{
    "category": "exampleSWCL",
    "changes": "Initial release",
    "conflicts": [
        {
            "operator": ">=",
            "shortName": "SWCL_TEST1_VENDOR_B",
            "version": "0.0.0-0"
        }
    ],
    "depends": [
        {
```

```
            "operator": ">=",
            "shortName": "SWCL_BASE",
            "version": "1.0.0-0"
        }
    ],
    "diagnosticAddress": "17473",
    "license": "AUTOSAR",
    "shortName": "SWCL_TEST1_VENDOR_A",
    "typeApproval": "None",
    "uuid": "0925f048-1570-11e9-8000-4f62f9ef99ce",
    "vendorID": "1447315780",
    "version": "0.1.0-1547190872"
}
```

Currently no delta packages are implemented, i.e. every package replaces the existing software cluster during processing. For delta package type packages the install steps shall be applied on the existing software cluster on the NVM, i.e. they shall be merged.

5 VPM design aspects

This chapter details the design approach of the VPM implementation.

5.1 Class Overview

The VPM implementation is following the modeling of the VehiclePackage of AUTOSAR_TPS_ManifestSpecification.

The VehiclePackageManager receives a VehiclePackage, parses it and generates a structure of RolloutSteps. These contain several UcmStep which themselves contain SwpSteps. The campaign is then processed according to this structure.

The VPM state machine is implemented by the class VpmStateMachine which is based on the Boost StateChart Library.

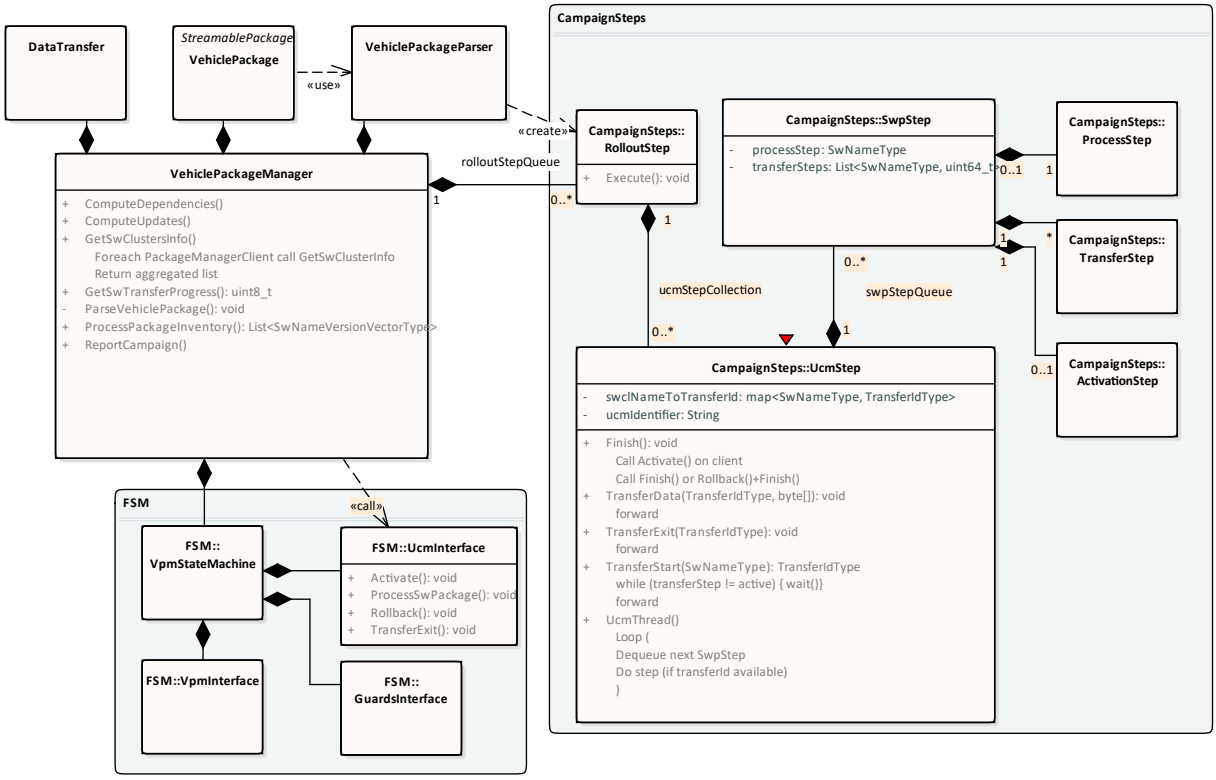

Figure 5.1: Vehicle Package Manager - class diagram

6 VPM implementation details

This place is reserved for future content.

7 UCM library functions

UCM library provides package receive and extraction functions.

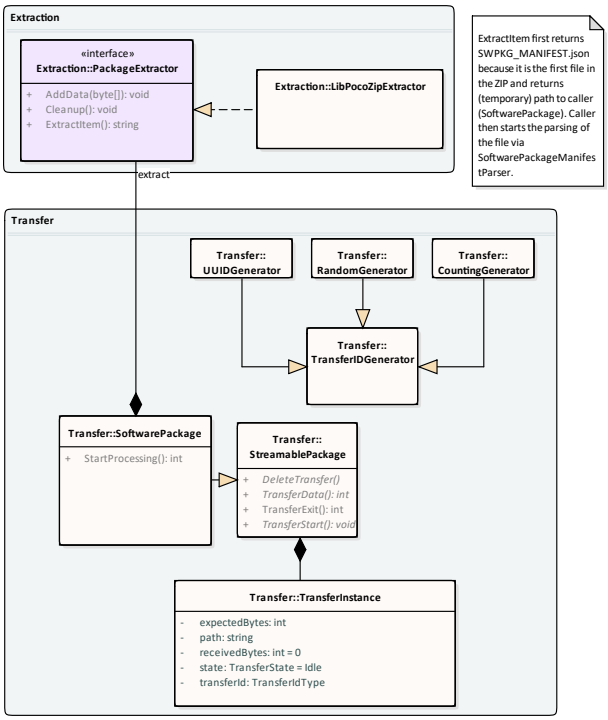

Figure 7.1: UCM library

7.1 Transfer

The following figure shows the receive component overview.

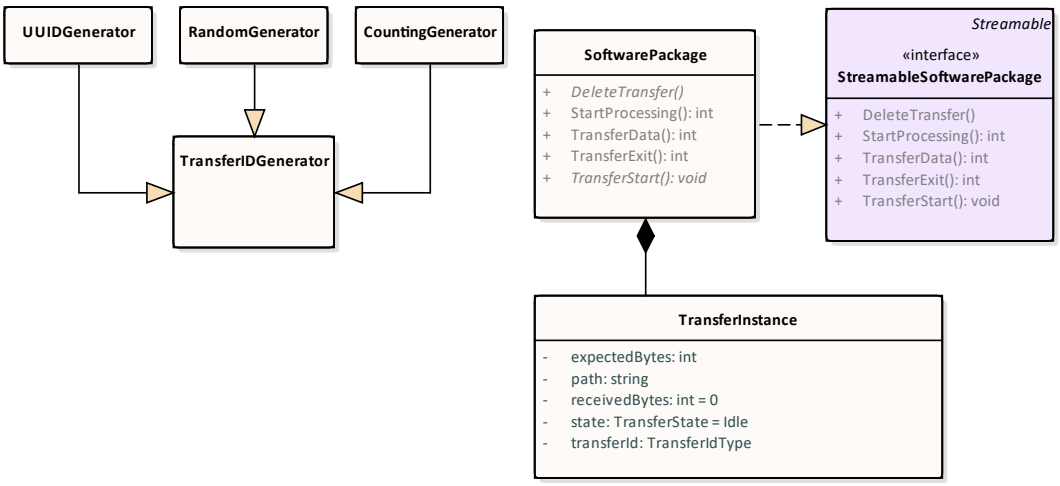

Figure 7.2: Receive component

7.1.1 Streamable software package

The streamable software package covers the traditional transmission of an update package to an ECU via UDS diagnostics using the UCM service methods Transfer Start, TransferData and TransferExit.

The service shall support concurrent uploads by multiple clients, therefore the design is separated into multiple classes and interfaces. The incoming service interface request is received by the package manager and forwarded to an individual object instance of StreamableSoftwarePackage retrieved from SynchronizedStorage which maps package objects by transfer ids.

Upon receiving a TransferStart, the SynchronizedStorage generates a new Transfer Id, instantiates a new StreamableSoftwarePackage object and maps both in a data structure. It then calls TransferStart on the new object with parameter size forwarded from the original request. The TransferId is hidden from the instance object and only used inside the SynchronizedStorage to map subsequent incoming calls to Transfer Data and TransferExit to the related instance object. The size value is stored inside the instance object and it opens a file handle for storing the data. The transfer state is set to Transferring as shown in the following picture and the function returns. The PackageManager's TransferStart then creates the client return value by combining the generated TransferId with the result of the object instance call TransferStartSuccess Type to a TransferStartReturnType.

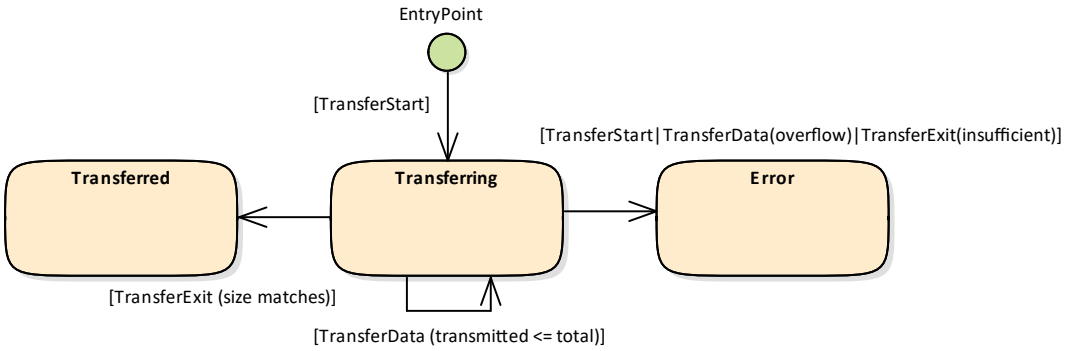

Figure 7.3: Transition of internal transfer states

If TransferData or TransferExit is now called at PackageManager or VPM, it looks up the instance object associated with the provided TransferId, calls the objects Transfer Data or TransferExit methods and returns the objects result to the client. If no instance object is found, an error is directly returned to the client, therefore this transition is not included in the instance state machine.

A TransferInstance object in state Transferring has to keep track of and compare size of the incoming data in TransferData to the expected size set on TransferStart. It has to detect overflow of data as well as a TransferExit without enough data transmitted before. If the size matches it will move to Transferred state, otherwise to Error state.

8 Demonstrator tooling

8.1 Package generation tooling

8.1.1 Usage from CMake

A CMake module is provided to help generating a software package for the APD. The module code is located in ara-api (`../../../../ara-api/ucm/ara-ucm-native/files/cmake/ara-ucm.cmake`). It provides the function add_software_package(), the following code snippet provides an example use from swcl_sample. In the top level CMakeLists (`../../../../sample-applications/ucm_examples/swcl_sample/CMakeLists.txt`) the macro is imported:

```
# ara::ucm macros
find_package(ara-ucm REQUIRED)
```

In the src level CMakeLists (`../../../../sample-applications/ucm_examples/swcl_sample/src/CMakeLists.txt`) the macro is used like this:

```
add_software_package(
    SWP_TARGET ${APP_EXECUTABLE}
    PACKAGE_NAME ${APP_NAME}
    PACKAGE_VERSION "1.0.0"
    ACTION_TYPE "Install"
    APP_MANIFEST "${CMAKE_SOURCE_DIR}/files/MANIFEST.json"
)
```

It packages everything that `make install` creates. Build with these commands

```
source /opt/apd/R20-11/environment-setup-i586-poky-linux
mkdir -p ~/AUTOSAR/build
cd ~/AUTOSAR/build/
cmake ~/AUTOSAR/sample-applications/ucm_examples/swcl_sample/
make
```

Calling make in the build directory generates the project and also the software package. Output should be something like this:

```
-- Toolchain file defaulted to '/opt/apd/R20-11/sysroots/x86_64-pokysdk-linux/usr/share/cmake/OEToolchainConfig.cmake'
-- The CXX compiler identification is GNU 8.3.0
-- Check for working CXX compiler: /opt/apd/R20-11/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/i586-poky-linux-g++
-- Check for working CXX compiler: /opt/apd/R20-11/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/i586-poky-linux-g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- option ARA_RUN_TESTS=OFF
-- option ARA_ENABLE_DEBUG=OFF (depends on ARA_RUN_TESTS)
-- Importing Dependencies
-- Boost version: 1.69.0
-- Looking for C++ include pthread.h
```

```
-- Looking for C++ include pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- Found Threads: TRUE
CMake Warning at /opt/apd/R20-11/sysroots/i586-poky-linux/usr/share/ara-ucm
    /ara-ucm.cmake:71 (message):
  add_software_package: Missing or invalid APP_PER
Call Stack (most recent call first):
  src/CMakeLists.txt:45 (add_software_package)


-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/AUTOSAR/build

Scanning dependencies of target swcl_sample
[ 33%] Building CXX object src/CMakeFiles/swcl_sample.dir/main.cpp.o
[ 66%] Linking CXX executable swcl_sample
[ 66%] Built target swcl_sample
Scanning dependencies of target swp
[100%] Generating swcl_sample_1.0.0_Install.zip
Creating swcl_sample_1.0.0_Install.zip
  adding: SWPKG_MANIFEST.json (deflated 43%)
  adding: SWCL_MANIFEST.json (deflated 47%)
  adding: swcl_sample/ (stored 0%)
  adding: swcl_sample/bin/ (stored 0%)
  adding: swcl_sample/bin/swcl_sample (deflated 65%)
  adding: swcl_sample/etc/ (stored 0%)
  adding: swcl_sample/etc/MANIFEST.json (deflated 60%)
  adding: swcl_sample/var/ (stored 0%)
[100%] Built target swp
```

In the above case the Software Package is then created as `~/AUTOSAR/build/src/swcl_sample_1.0.0_Install.zip`.

8.1.2 Usage from bitbake

CMake projects including the above snippets automatically create a SWP in their bitbake build directory as it is part of the `make all` instruction.

The SWP zip file is copied to the deploy directory by the `do_deploy` step in SWPKG class (`../../../../yocto-layers/meta-ara/classes/apd_ucm_swpkg.bbclass`). An example can be seen in recipe ucm-swcl-sample.bb (`../../../../yocto-layers/meta-ara/recipes-ara/sample-applications/ucm-swcl-sample.bb`) which inherits the class and thus copies the SWP archive to `build/tmp/deploy/images/qemux86/autosar-swp/ucm-swcl-sample-1.0` in the deploy step.