Document Title	Demonstrator Design of Functional Cluster Execution Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	843

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	• Updated known limitations • Minor editorial changes
2019-11-28	R19-11	AUTOSAR Release Management	• Updated known limitations. • Removed generated API tables. • Changed Document Status from Final to published.
2019-03-29	19-03	AUTOSAR Release Management	• STL types are replaced with ara::core types. • Added a test for the execution dependencies. • Tests are enabled for running on the CI server. • Updated the diagrams.
2018-11-02	18-10	AUTOSAR Release Management	• Added Function group implementation. • Added Resource group implementation. • Updated known limitations.
2018-05-02	18-03	AUTOSAR Release Management	• Generated figures in chapter 2 from code. • Fixed editorial and content issues.
2017-11-22	17-10	AUTOSAR Release Management	• Minor changes

2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Table of Contents

1 Introduction

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

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply to 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 Execution 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 proofs of concept for Execution Management features specified in the Execution Management specification of AUTOSAR Adaptive Platform. 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 from this implementation, will have to further satisfy the safety constraints described in the industry standards.

1.1 Known limitations

The Execution Management implementation has many limitations. This chapter provides a documentation of all specifications and their current implementation status, also if functionality is not yet there or still deviates from the specifications.

1.1.1 Overview

- Machine and Execution Manifests are not yet fully utilized by the code.
  - Basically, the Execution Management searches for the manifest files relatively to some root directory, which is by default the system root path ("/").
  - The root directory for debugging might be re-defined by passing the *ARA_ROOT=<new-root-path>* command line parameter to the Execution Management.
  - The Execution Management searches *ARA_ROOT/opt/<application-name>/bin* and *ARA_ROOT/opt/<application-name>/etc* directories for binary and manifest files of an application accordingly.
  - The Machine Manifest is searched in the *ARA_ROOT/etc/system/machine_manifest.json* file, whose path is currently hard-coded.

  - Currently, all the manifests, including Execution Manifests and a Machine Manifest, are searched once by Execution Management at its startup stage.

- Current realization of the Execution Management does not scan for updated software. If a software package has been updated by the Update and Configuration Management, it simply performs a reset for rebooting a machine.

1.1.2 Dependencies on other Functional Clusters

- Persistency
- Log and Trace

1.1.3 Limitations for chapter "Execution Management Responsibilities"

Spec item	Status
SWS_EM_01030 Restriction of process creation right for Processes	not implemented

1.1.4 Limitations for chapter "Process Lifecycle Management / Execution State"

Spec item	Status
SWS_EM_01055 Initiation of Process termination	implemented
SWS_EM_01314 Default value for terminationBehaviour	not implemented
SWS_EM_01309 Unexpected Termination of a process	not implemented
SWS_EM_02243 Handling Execution State Running	not implemented
SWS_EM_01402 Implicit Running Process State	partially implemented
SWS_EM_01403 Reporting Non-reporting Process	not implemented

1.1.5 Limitations for chapter "Process Lifecycle Management / Process States"

Spec item	Status
SWS_EM_01401 Process Self Reporting	implemented
SWS_EM_01002 Idle Process State	implemented
SWS_EM_01003 Starting Process State	implemented
SWS_EM_01004 Running Process State of Reporting Processes	implemented
SWS_EM_01404 Terminating Process State after Termination Request	implemented
SWS_EM_01006 Terminated Process State	implemented

1.1.6 Limitations for chapter "Process Lifecycle Management / Startup and Termination"

Spec item	Status
SWS_EM_01050 Start Dependent Processes	implemented
SWS_EM_01051 Termination of Processes	implemented
SWS_EM_01001 Execution Dependency error	not implemented
SWS_EM_01012 Process Argument Passing	implemented
SWS_EM_01072 Process Argument Zero	implemented
SWS_EM_01073 Simple Arguments (obsolete)	not implemented
SWS_EM_01074 Short form arguments with option value (obsolete)	not implemented
SWS_EM_01075 Short form Arguments without option value (obsolete)	not implemented
SWS_EM_01076 Long form Arguments with option value (obsolete)	not implemented
SWS_EM_01077 Long form Arguments without option value (obsolete)	not implemented
SWS_EM_01078 Process Arguments strings	implemented
SWS_EM_02246 Process specific Environment Variables	implemented
SWS_EM_02247 Machine specific Environment Variables	not implemented
SWS_EM_02249 Missing value from Environment Variable definition	not implemented
SWS_EM_02248 Environment Variables precedence	not implemented

1.1.7 Limitations for chapter "Process Lifecycle Management / Startup Sequence"

Spec item	Status
SWS_EM_01000 Startup order	implemented

1.1.8 Limitations for chapter "State Management"

Spec item	Status
SWS_EM_01032 Machine States configuration	implemented with deviations
SWS_EM_02250 Machine State Startup	implemented
SWS_EM_01023 Self initiation of Machine State Startup transition	implemented
SWS_EM_02241 Machine State Startup Completion	not implemented
SWS_EM_01107 Function Group configuration	implemented
SWS_EM_01013 Function Group State	implemented
SWS_EM_01033 Process start-up configuration	implemented
SWS_EM_01109 Misconfigured Process - not assigned to a Function Group	not implemented
SWS_EM_02254 Misconfigured Process - assigned to more than one Function Group	not implemented
SWS_EM_01110 Off States	implemented

▽

△

SWS_EM_01060 State transition - termination behavior	implemented
SWS_EM_02251 State transition - restart behavior	implemented
SWS_EM_01065 State transition - Process termination timeout monitoring	implemented
SWS_EM_02255 State transition - Process termination timeout reaction	implemented
SWS_EM_02258 State transition - Process termination timeout reporting	not implemented
SWS_EM_02311 Order of process termination timeout reaction	not implemented
SWS_EM_01066 State transition - start behavior	implemented
SWS_EM_02253 State transition - Process start-up timeout monitoring	implemented
SWS_EM_02260 State transition - Process start-up timeout reaction	not implemented
SWS_EM_02280 Effect on Execution Dependecy	not implemented
SWS_EM_02310 State transition - process termination after start-up timeout reaction	not implemented
SWS_EM_02259 State transition - Process start-up timeout reporting	not implemented
SWS_EM_02312 Order of process start-up timeout reaction	not implemented
SWS_EM_02245 Dependency resolution during state change	implemented with deviations
SWS_EM_01067 Actions on complete state transition	implemented with deviations
SWS_EM_02313 Unexpected Termination of a starting process during Function Group State transition	not implemented
SWS_EM_02314 Unexpected Termination of terminating processes during Function Group State transition	not implemented
SWS_EM_02297 StateClient usage restriction	not implemented

1.1.9 Limitations for chapter "Deterministic Execution"

The entire chapter is not implemented.

Spec item	Status
SWS_EM_01301 Cyclic Execution	not implemented
SWS_EM_01302 Cyclic Execution Control	not implemented
SWS_EM_01303 Cyclic Execution Control Sequence	not implemented
SWS_EM_01304 Service Modification	not implemented
SWS_EM_01351 Execution Cycle Time	not implemented
SWS_EM_01352 Execution Cycle Timeout	not implemented
SWS_EM_01353 Event-triggered Cycle Activation	not implemented
SWS_EM_01305 Worker Pool	not implemented
SWS_EM_01306 Processing Container Objects	not implemented
SWS_EM_01307 Worker Object	not implemented
SWS_EM_01308 Random Numbers	not implemented

▽

△

SWS_EM_01310 Get Activation Time	not implemented
SWS_EM_01311 Activation Time Unknown	not implemented
SWS_EM_01312 Get Next Activation Time	not implemented
SWS_EM_01313 Next Activation Time Unknown	not implemented
SWS_EM_01320 Number of DeterministicClients	not implemented
SWS_EM_01321 Minimum number of required synchronization requests	not implemented
SWS_EM_01322 Calculation of the next cycle	not implemented
SWS_EM_01323 Total kRun loop count	not implemented
SWS_EM_01324 Infinite kRun loop	not implemented
SWS_EM_01325 Synchronization Request Message	not implemented
SWS_EM_01326 Synchronization Response Message	not implemented
SWS_EM_01327 Return of the wait point API	not implemented

1.1.10 Limitations for chapter "Resource Limitation"

Spec item	Status
SWS_EM_02102 Memory control	implemented
SWS_EM_02103 CPU usage control	implemented
SWS_EM_02104 Core affinity	not implemented
SWS_EM_01014 Scheduling policy	implemented
SWS_EM_01041 Scheduling FIFO	implemented
SWS_EM_01042 Scheduling Round-Robin	implemented
SWS_EM_01043 Scheduling Other	implemented
SWS_EM_01015 Scheduling priority	implemented
SWS_EM_02106 ResourceGroup assignment	implemented
SWS_EM_02107 Maximum heap	not implemented
SWS_EM_02108 Maximum system memory usage	not implemented
SWS_EM_02109 Process pre-mapping	not implemented

1.1.11 Limitations for chapter "Fault Tolerance"

The entire chapter is not implemented.

Spec item	Status
SWS_EM_02032 On entry to Unrecoverable state	not implemented
SWS_EM_02033 after execution of pre-cleanup action	not implemented
SWS_EM_02034 after all processes managed by EM terminated	not implemented

1.1.12 Limitations for chapter "Security / Trusted Platform"

The entire chapter is not implemented.

Spec item	Status
SWS_EM_02299 Availability of a Trust Anchor	not implemented
SWS_EM_02300 Integrity and Authenticity of processed Machine Manifest	not implemented
SWS_EM_02301 Integrity and Authenticity of each Executable	not implemented
SWS_EM_02302 Integrity and Authenticity of shared objects	not implemented
SWS_EM_02303 Integrity and Authenticity of processed Execution Manifests	not implemented
SWS_EM_02304 Integrity and Authenticity of processed Service Instance Manifests	not implemented
SWS_EM_02305 Failed authenticity checks	not implemented
SWS_EM_02306 Machine Manifest	not implemented
SWS_EM_02307 Strict Mode - Execution manifest	not implemented
SWS_EM_02308 Strict Mode - Service Instance manifests	not implemented
SWS_EM_02309 Strict Mode - Executables	not implemented

1.1.13 Limitations for the public API / Type Definitions

Spec item	Status
SWS_EM_02000 ExecutionState	implemented
SWS_EM_02201 ActivationReturnType	not implemented
SWS_EM_02202 ActivationTimeStampReturnType	not implemented
SWS_EM_02541 Execution Error	not implemented
SWS_EM_02544 ExecutionErrorEvent	not implemented
SWS_EM_02545 ExecutionErrorEvent::executionError	not implemented
SWS_EM_02546 ExecutionErrorEvent::functionGroup	not implemented

1.1.14 Limitations for the public API / Class Definitions

Spec item	Status
SWS_EM_02001 ExecutionClient class	implemented
SWS_EM_02030 ExecutionClient::ExecutionClient	implemented
SWS_EM_02002 ExecutionClient::~ExecutionClient	implemented
SWS_EM_02003 ExecutionClient::ReportExecutionState	implemented with deviations
SWS_EM_02510 WorkerRunnable class	not implemented
SWS_EM_02520 WorkerRunnable::Run	not implemented
SWS_EM_02530 WorkerThread class	not implemented
SWS_EM_02531 WorkerThread::WorkerThread	not implemented
SWS_EM_02532 WorkerThread::~WorkerThread	not implemented

▽

△

SWS_EM_02540 WorkerThread::GetRandom	not implemented
SWS_EM_02210 DeterministicClient class	not implemented
SWS_EM_02211 DeterministicClient::DeterministicClient	not implemented
SWS_EM_02215 DeterministicClient::~DeterministicClient	not implemented
SWS_EM_02216 DeterministicClient::WaitForNextActivation	not implemented
SWS_EM_02220 DeterministicClient::RunWorkerPool	not implemented
SWS_EM_02225 DeterministicClient::GetRandom	not implemented
SWS_EM_02226 DeterministicClient::SetRandomSeed	not implemented
SWS_EM_02230 DeterministicClient::GetActivationTime	not implemented
SWS_EM_02235 DeterministicClient::GetNextActivation Time	not implemented
SWS_EM_02263 FunctionGroup class	implemented
SWS_EM_02264 FunctionGroup::Preconstruct	implemented
SWS_EM_02265 FunctionGroup::FunctionGroup	implemented
SWS_EM_02266 FunctionGroup::~FunctionGroup	implemented
SWS_EM_02267 FunctionGroup::operator==	implemented
SWS_EM_02268 FunctionGroup::operator!=	implemented
SWS_EM_02269 FunctionGroupState class	implemented
SWS_EM_02270 FunctionGroupState::Preconstruct	implemented
SWS_EM_02271 FunctionGroupState::FunctionGroupState	implemented
SWS_EM_02272 FunctionGroupState::~FunctionGroup State	implemented
SWS_EM_02273 FunctionGroupState::operator==	implemented
SWS_EM_02274 FunctionGroupState::operator!=	implemented
SWS_EM_02275 StateClient class	partially implemented
SWS_EM_02276 StateClient::StateClient	implemented
SWS_EM_02277 StateClient::~StateClient	implemented
SWS_EM_02278 StateClient::SetState	implemented
SWS_EM_02298 Canceling ongoing state transition	not implemented
SWS_EM_02279 StateClient::GetInitialMachineState TransitionResult	implemented with deviations
SWS_EM_02542 StateClient::GetExecutionError	not implemented
SWS_EM_02543 Default value for ExecutionError	not implemented

1.1.15 Limitations for the public API / Errors

Spec item	Status
SWS_EM_02281 Execution Management error codes	partially implemented
SWS_EM_02282 ExecException class	implemented
SWS_EM_02283 ExecException::ExecException	implemented
SWS_EM_02290 GetExecErrorDomain	implemented
SWS_EM_02291 MakeErrorCode	implemented
SWS_EM_02284 ExecErrorDomain class	implemented
SWS_EM_02286 ExecErrorDomain::ExecErrorDomain	implemented

▽

△

SWS_EM_02287 ExecErrorDomain::Name	implemented
SWS_EM_02288 ExecErrorDomain::Message	implemented
SWS_EM_02289 ExecErrorDomain::ThrowAsException	implemented

2 Overview on architecture

2.1 Design approach / Design principles

In order to offer all the features described in the Execution Management specification, the current ara::exec implementation provides the following Demonstrator artifacts:

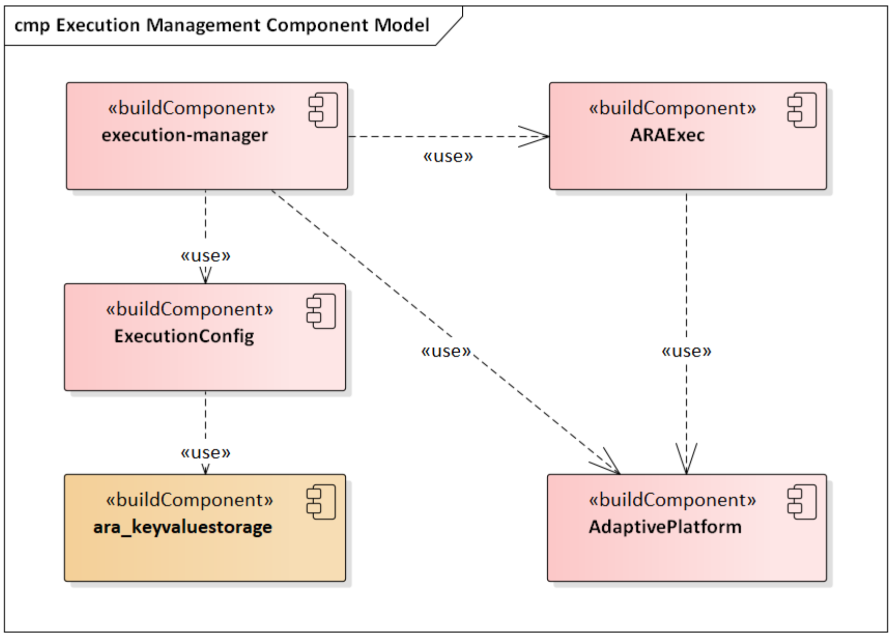
cmp Execution Management Component Model
«buildComponent» execution-manager
«buildComponent» ARAExec
«buildComponent» ExecutionConfig
«buildComponent» ara_keyvaluestorage
«buildComponent» AdaptivePlatform
«use»
«use»
«use»
«use»
«use»

Figure 2.1: Execution Management software - high level component diagram

- **execution-manager** is the software component responsible for platform lifecycle management and process lifecycle management.
- **ara_exec library** implements the specified public ara::exec API
- In addition to the above mentioned main deliverables, auxiliary libraries are developed:
  - **ExecutionConfig library** provides access to the content of the Machine Manifest and the Execution Manifests.
  - **AdaptivePlatform library** provides communication mechanisms for execution-manager.

2.2 Class Overview

2.2.1 Execution Manager

The Execution Manager is the main deliverable provided by the Execution Management functional cluster. It implements most of the features described in the Specification of Execution Management. It assumes the role of the first process in an AUTOSAR Adaptive Platform System, which is executed after the operating system startup and therefore is the parent of all other processes including platform-level application processes and user-level application processes.

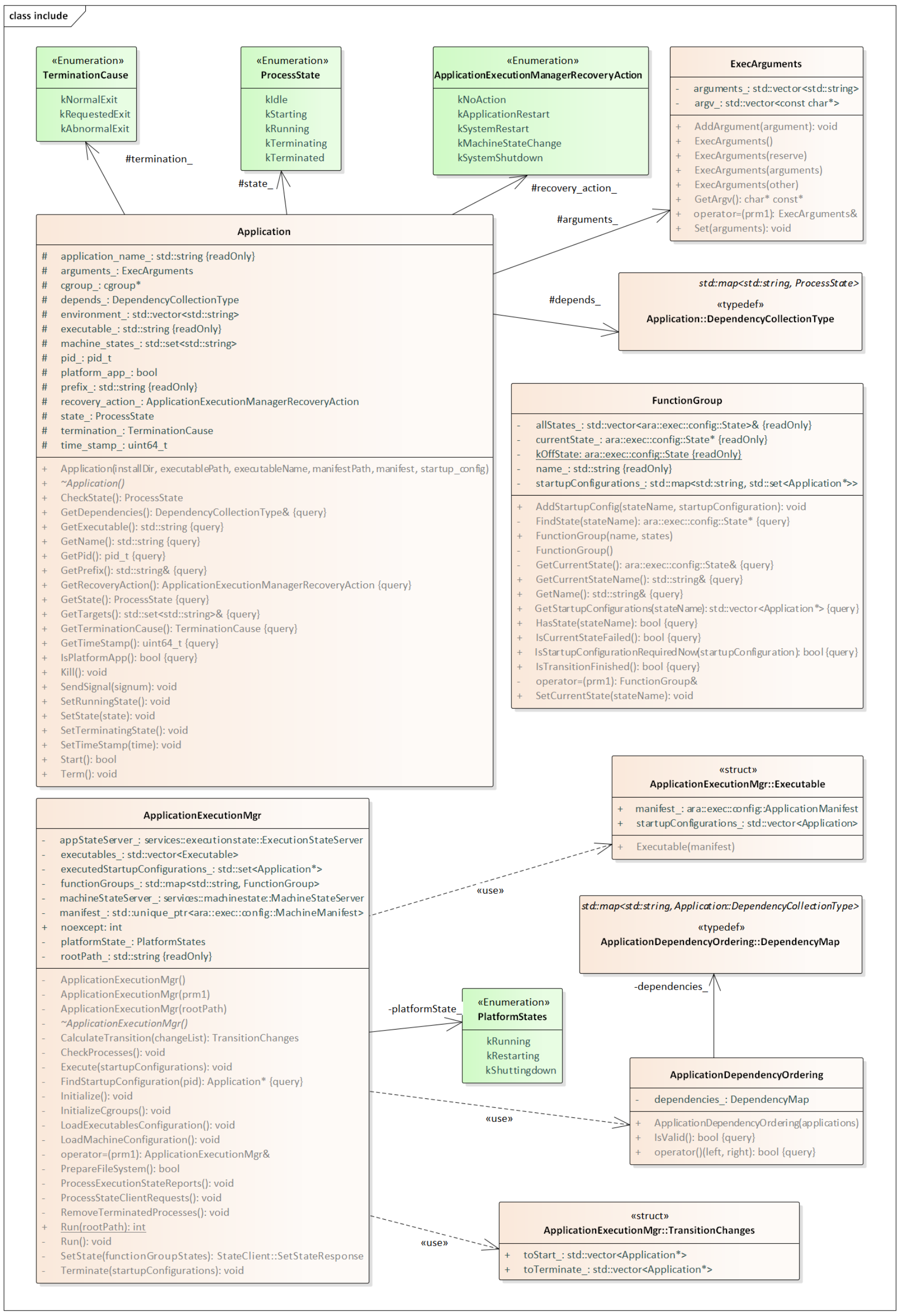
class include
«Enumeration»
TerminationCause
kNormalExit
kRequestedExit
kAbnormalExit
«Enumeration»
ProcessState
kIdle
kStarting
kRunning
kTerminating
kTerminated
«Enumeration»
ApplicationExecutionManagerRecoveryAction
kNoAction
kApplicationRestart
kSystemRestart
kMachineStateChange
kSystemShutdown
ExecArguments
- arguments_: std::vector<std::string>
- argv_: std::vector<const char*>
+ AddArgument(argument): void
+ ExecArguments()
+ ExecArguments(reserve)
+ ExecArguments(arguments)
+ ExecArguments(other)
+ GetArgv(): char* const*
+ operator=(prm1): ExecArguments&
+ Set(arguments): void
#termination_
#state_
#recovery_action_
#arguments_
#depends_
Application
# application_name_: std::string {readOnly}
# arguments_: ExecArguments
# cgroup_: cgroup*
# depends_: DependencyCollectionType
# environment_: std::vector<std::string>
# executable_: std::string {readOnly}
# machine_states_: std::set<std::string>
# pid_: pid_t
# platform_app_: bool
# prefix_: std::string {readOnly}
# recovery_action_: ApplicationExecutionManagerRecoveryAction
# state_: ProcessState
# termination_: TerminationCause
# time_stamp_: uint64_t
+ Application(installDir, executablePath, executableName, manifestPath, manifest, startup_config)
+ ~Application()
+ CheckState(): ProcessState
+ GetDependencies(): DependencyCollectionType& {query}
+ GetExecutable(): std::string {query}
+ GetName(): std::string {query}
+ GetPid(): pid_t {query}
+ GetPrefix(): std::string& {query}
+ GetRecoveryAction(): ApplicationExecutionManagerRecoveryAction {query}
+ GetState(): ProcessState {query}
+ GetTargets(): std::set<std::string>& {query}
+ GetTerminationCause(): TerminationCause {query}
+ GetTimeStamp(): uint64_t {query}
+ IsPlatformApp(): bool {query}
+ Kill(): void
+ SendSignal(signum): void
+ SetRunningState(): void
+ SetState(state): void
+ SetTerminatingState(): void
+ SetTimeStamp(time): void
+ Start(): bool
+ Term(): void
std::map<std::string, ProcessState>
«typedef»
Application::DependencyCollectionType
FunctionGroup
- allStates_: std::vector<ara::exec::config::State>& {readOnly}
- currentState_: ara::exec::config::State* {readOnly}
- kOffState: ara::exec::config::State {readOnly}
- name_: std::string {readOnly}
- startupConfigurations_: std::map<std::string, std::set<Application*>>
+ AddStartupConfig(stateName, startupConfiguration): void
- FindState(stateName): ara::exec::config::State* {query}
+ FunctionGroup(name, states)
- FunctionGroup()
- GetCurrentState(): ara::exec::config::State& {query}
+ GetCurrentStateName(): std::string& {query}
+ GetName(): std::string& {query}
+ GetStartupConfigurations(stateName): std::vector<Application*> {query}
+ HasState(stateName): bool {query}
+ IsCurrentStateFailed(): bool {query}
+ IsStartupConfigurationRequiredNow(startupConfiguration): bool {query}
+ IsTransitionFinished(): bool {query}
- operator=(prm1): FunctionGroup&
+ SetCurrentState(stateName): void
«struct»
ApplicationExecutionMgr::Executable
+ manifest_: ara::exec::config::ApplicationManifest
+ startupConfigurations_: std::vector<Application>
+ Executable(manifest)
ApplicationExecutionMgr
- appStateServer_: services::executionstate::ExecutionStateServer
- executables_: std::vector<Executable>
- executedStartupConfigurations_: std::set<Application*>
- functionGroups_: std::map<std::string, FunctionGroup>
- machineStateServer_: services::machinestate::MachineStateServer
- manifest_: std::unique_ptr<ara::exec::config::MachineManifest>
+ noexcept: int
- platformState_: PlatformStates
- rootPath_: std::string {readOnly}
- ApplicationExecutionMgr()
- ApplicationExecutionMgr(prm1)
- ApplicationExecutionMgr(rootPath)
- ~ApplicationExecutionMgr()
- CalculateTransition(changeList): TransitionChanges
- CheckProcesses(): void
- Execute(startupConfigurations): void
- FindStartupConfiguration(pid): Application* {query}
- Initialize(): void
- InitializeCgroups(): void
- LoadExecutablesConfiguration(): void
- LoadMachineConfiguration(): void
- operator=(prm1): ApplicationExecutionMgr&
- PrepareFileSystem(): bool
- ProcessExecutionStateReports(): void
- ProcessStateClientRequests(): void
- RemoveTerminatedProcesses(): void
+ Run(rootPath): int
- Run(): void
- SetState(functionGroupStates): StateClient::SetStateResponse
- Terminate(startupConfigurations): void
«use»
std::map<std::string, Application::DependencyCollectionType>
«typedef»
ApplicationDependencyOrdering::DependencyMap
-dependencies_
-platformState_
«Enumeration»
PlatformStates
kRunning
kRestarting
kShuttingdown
ApplicationDependencyOrdering
- dependencies_: DependencyMap
+ ApplicationDependencyOrdering(applications)
+ IsValid(): bool {query}
+ operator()(left, right): bool {query}
«use»
«use»
«struct»
ApplicationExecutionMgr::TransitionChanges
+ toStart_: std::vector<Application*>
+ toTerminate_: std::vector<Application*>

Figure 2.2: Execution Manager - class diagram

The Demonstrator implementation of Adaptive Platform is built on top of Linux, having the applications run temporally and spatially separated. One of the key aspects, which has been stressed during the Adaptive Platform standardization process, was the opportunity of having the applications running isolated from each other, being free from any type of interferences. The POSIX processes and its associated mechanisms offered by Linux fit perfectly into this picture. The following POSIX mechanisms are used to implement a Process lifecycle of an Adaptive Application.

First, a combination of the *fork()/execv()* POSIX calls is used to execute a Process of an Adaptive Application. The *fork()* function will create a new child process derived from the Execution Management process. A call to *execv()* replaces the derived process image with a new image of the creating process of the application. For illustration purposes, the implementation of starting a Process is shown below.

```
bool Application::Start()
{
    if (pid_ != 0) {
        TRACE_ERROR("EM: Invalid call for Start(): the process is already
running");
        return false;
    }

    TRACE_INFO("EM: Starting executable " << executable_);

    int pid = ::fork();
    if (pid < 0) {
        TRACE_ERROR("EM: fork() failed"
            << ", errno: "
            << errno);
        return false;
    }

    if (pid == 0) {
        // The child process

        const char* childWorkDir = GetPrefix().c_str();

        // Change working directory to application root
        if (::chdir(childWorkDir) != 0) {
            TRACE_FATAL("EM: chdir() failed for dir " << childWorkDir << ",
errno: " << errno);
        } else {
            // Set environment variables
            for (const auto& variable : environment_) {
                ::putenv(const_cast<char*>(variable.c_str()));
            }

            // Redirect terminal output for application to /var/redirected
/<application_name_>
            utility::RedirectProcessOutput(application_name_.c_str());

            const char* childPath = executable_.c_str();
            char* const* childArgv = const_cast<char* const*>(arguments_.
GetArgv());
```

```

            // Execute the executable with the specified arguments
            ::execv(childPath, childArgv);

            // When execv() is successful, the current process is replaced by the child.
            // Otherwise, the following code will be reached.
            TRACE_FATAL("EM: execv() failed for executable " << childPath << ", errno: " << errno);
        }

        // Terminate the failed child process
        std::abort();
        return false;
    } else if (pid > 0) {
        // The parent process

        state_ = ProcessState::kStarting;
        pid_ = pid;

        TRACE_INFO("EM: Forked child '" << executable_ << "' with PID " << pid_);

        // The only successfull return
        return true;
    }
}
```

Secondly, process termination is implemented by sending the *SIGTERM* or *SIGKILL* signal to it. An assumption is that all Processes shall run in a cooperative fashion, which means they shall react on receiving the *SIGTERM* signal and properly exit in a certain period of time. If this does not happen in the specified term by any reason, the *SIGKILL* signal will be sent in order to forcibly terminate the Process.

2.2.2 ara_exec library

This library comprises the public API which the Execution Management functional cluster exposes to the processes. It consists of two kind of interfaces:

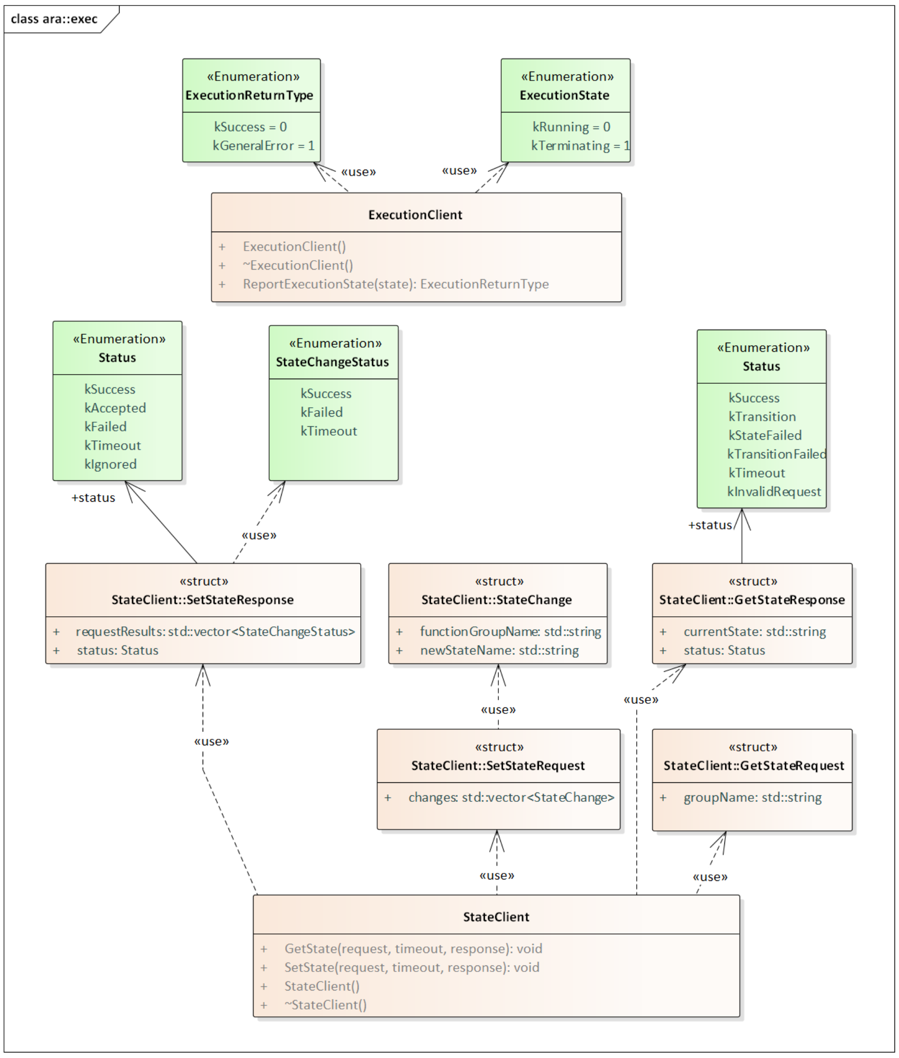

Figure 2.3: ara::exec - class diagram

ExecutionClient Interface - the idea of having an interface exposed to all processes where they can report their execution state has come into the picture once we understood it is not enough to fork in a number of new processes and just assume that the new state is immediately installed on the platform. It is in fact up to the application developer to judge when his process is fully initialized such that it can switch to the so called "Running" execution state, which will be considered for confirming the new state to the machine. Therefore, we realized the necessity of getting some sort of feedback back to Execution Management, regarding the actual execution state from within each executed application. This mechanism grants a higher degree of confidence when it comes to confirming the new state. Such features might play a crucial role for safety relevant systems where the confirmation received from all spawned applications may provide a good indicator that all the prerequisites have been created for entering a safety critical mode of operation.

StateClient Interface - exposed to an exclusive group of processes which are authorized to request a new ECU state. When we talk about state for the Adaptive Platform, we talk about modes of operation for an ECU. The goal here was to avoid nesting of multiple levels of states. We came to the conclusion that a single state level would be enough to achieve the goals of State Management on Adaptive Platform. One of the implications that has arisen immediately was the necessity of standardizing a number of Machine States in order to cover the use cases related to platform initialization and platform shutting down.

2.2.3 ExecutionConfig library

Execution Management is most likely the first client of the Key Value Storage feature belonging to the Persistency functional cluster on the Adaptive Platform. Since this feature exposes a generic API offering access to the persistent data for the typical types used today in C++ programming, we decided to introduce a convenience layer on top of that to handle the specifics of Application Manifest and Machine Manifest contents.

Side note: for the current implementation of the Demonstrator, this library is released for Execution Management private use. However, it is expected that for future releases, these interfaces will become public within the Adaptive Platform once that necessity arises.

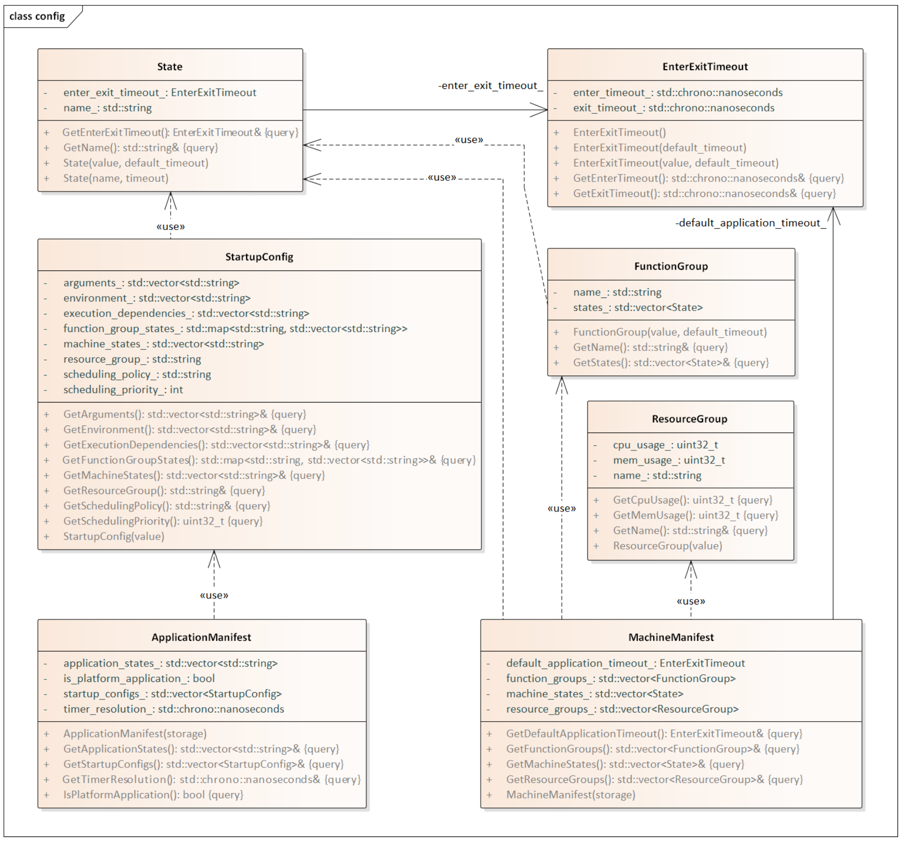
class config
State
- enter_exit_timeout_: EnterExitTimeout
- name_: std::string
+ GetEnterExitTimeout(): EnterExitTimeout& {query}
+ GetName(): std::string& {query}
+ State(value, default_timeout)
+ State(name, timeout)
-enter_exit_timeout_
EnterExitTimeout
- enter_timeout_: std::chrono::nanoseconds
- exit_timeout_: std::chrono::nanoseconds
+ EnterExitTimeout()
+ EnterExitTimeout(default_timeout)
+ EnterExitTimeout(value, default_timeout)
+ GetEnterTimeout(): std::chrono::nanoseconds& {query}
+ GetExitTimeout(): std::chrono::nanoseconds& {query}
«use»
«use»
«use»
-default_application_timeout_
StartupConfig
- arguments_: std::vector<std::string>
- environment_: std::vector<std::string>
- execution_dependencies_: std::vector<std::string>
- function_group_states_: std::map<std::string, std::vector<std::string>>
- machine_states_: std::vector<std::string>
- resource_group_: std::string
- scheduling_policy_: std::string
- scheduling_priority_: int
+ GetArguments(): std::vector<std::string>& {query}
+ GetEnvironment(): std::vector<std::string>& {query}
+ GetExecutionDependencies(): std::vector<std::string>& {query}
+ GetFunctionGroupStates(): std::map<std::string, std::vector<std::string>>& {query}
+ GetMachineStates(): std::vector<std::string>& {query}
+ GetResourceGroup(): std::string& {query}
+ GetSchedulingPolicy(): std::string& {query}
+ GetSchedulingPriority(): uint32_t {query}
+ StartupConfig(value)
FunctionGroup
- name_: std::string
- states_: std::vector<State>
+ FunctionGroup(value, default_timeout)
+ GetName(): std::string& {query}
+ GetStates(): std::vector<State>& {query}
ResourceGroup
- cpu_usage_: uint32_t
- mem_usage_: uint32_t
- name_: std::string
+ GetCpuUsage(): uint32_t {query}
+ GetMemUsage(): uint32_t {query}
+ GetName(): std::string& {query}
+ ResourceGroup(value)
«use»
«use»
«use»
ApplicationManifest
- application_states_: std::vector<std::string>
- is_platform_application_: bool
- startup_configs_: std::vector<StartupConfig>
- timer_resolution_: std::chrono::nanoseconds
+ ApplicationManifest(storage)
+ GetApplicationStates(): std::vector<std::string>& {query}
+ GetStartupConfigs(): std::vector<StartupConfig>& {query}
+ GetTimerResolution(): std::chrono::nanoseconds& {query}
+ IsPlatformApplication(): bool {query}
MachineManifest
- default_application_timeout_: EnterExitTimeout
- function_groups_: std::vector<FunctionGroup>
- machine_states_: std::vector<State>
- resource_groups_: std::vector<ResourceGroup>
+ GetDefaultApplicationTimeout(): EnterExitTimeout& {query}
+ GetFunctionGroups(): std::vector<FunctionGroup>& {query}
+ GetMachineStates(): std::vector<State>& {query}
+ GetResourceGroups(): std::vector<ResourceGroup>& {query}
+ MachineManifest(storage)

Figure 2.4: ExecutionConfig - class diagram

2.3 Start-up sequence overview

The Execution Management is started first by the operating system. The *main()* function of Execution Management:

1. Checks passed parameters to the *main()* function and looks for a new path for the *ARA_ROOT* root directory.
2. Builds a main object, initializes variables of the object, reads the root directory, and parses the Machine Manifest and Execution Manifests of Adaptive Applications saving their parameters to the object's internal data structures. Note that this step happens only once per lifecycle of the Execution Management process.
3. Sets the mandatory state "Startup" of the Function Group "Machine".
4. Starts Processes of Adaptive Applications which reference to the Machine State Startup.
5. Executes a main internal loop while an internal flag of the object is set to Running.

In the main internal loop, Execution Management basically listens to requesters of state transitions. Having received a request to change the current state of a Function Group, Execution Management begins to manage Processes.

1. If a Process references to one state of a Function Group, the process will be executed only when the state is requested and terminated when the state is shutdown.
2. If a Process references to different states of a Function Group, or a few Function Groups and has one startup configuration for the states, the process will be left running while state transition between the states.
3. If a Process references to different states of a Function Group and has different startup configurations for the states, the process will be terminated in its current configuration and restarted in a new configuration while state transition.

2.4 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
-GTest/GMock-	-1.7.0-	- Unit Testing-	- static library-	-BSD-	-NA-