The Framework Profile
C1 Implementation
- USER REQUIREMENTS -

Alessandro Pasetti & Vaclav Cechticky

13 October 2016

Revision 1.2.2
PP-SP-COR-0001

P&P Software GmbH
High Tech Center 1
8274 Tägerwilen
Switzerland

Web site: `www.pnp-software.com`
E-mail: `pnp-software@pnp-software.com`

Abstract

This document defines, justifies, and verifies the User Requirements for the C1 Implementation of the FW Profile. The FW Profile is a specification-level modelling language defined as a restriction of UML. The core modelling constructs offered by the FW Profile are State Machines, Procedures (equivalent to UML's Activity Diagrams), and RT Containers (encapsulations of threads).

The FW Profile is implementation-independent. The C1 Implementation is a C language implementation of the modelling concepts of the FW Profile. The main features of the C1 Implementation are: small memory footprint, small CPU demands, scalability, and high reliability.

The C1 Implementation is provided with a Qualification Data Package which can be used to support the certification of applications built using its components.

Contents

List of Figures

List of Tables

Listings

1 Change History

This section lists the changes made in successive revisions of this document. Changes are classified according to their type. The change type is identified in the second column in the table according to the following convention:

- "**E**": Editorial or stylistic change
- "**L**": Clarification of existing text
- "**D**": A requirement or part of a requirement which was present in the previous revision has been deleted
- "**C**": A requirement or part of a requirement which was presented in the previous revision has been changed
- "**N**": A new requirement has been introduced

Table 1: Changes introduced in Revision 1.2.2

Section	Type	Description
n.a.	E	Corrected reference number in [2]

Table 2: Changes introduced in Revision 1.2.1

Section	Type	Description
n.a.	E	Corrected error in document reference number in page headers

Table 3: Changes introduced in Revision 1.2.0

Section	Type	Description
n.a.	L	Clarified formulation of abstract
n.a.	L	Added list of code listings to the table of contents
2	L	Extended introduction to cover RT Containers
2.1	E	Minor editorial corrections
3.3	E	Minor editorial change in verification part of requirement FW-3.3.2; in requirement FW-3.3.3, corrected reference to: "`FwSmTestCaseCheck7`", to reference to: "`FwSmTestCaseCheck6`"; in requirement FW-3.3.5: corrected reference to section 5.2 to reference to section 4.2
3.3	N	Added a note to requirement FW-3.3.2 to clarify the definition of "configuration" and modified requirement justification to reflect this definition
3.4	E	In requirement FW-3.4.3: changed "`FwSmStart`" to "`FwSmStop`"
3.4	E	Minor editorial change in implementation part of requirement FW-3.4.2
3.5	E	Minor editorial correction in justification part of requirement FW-3.5.2; in requirement FW-3.5.5: changed reference to section 5.2, to reference to section 4.2; fixed typo in verification part of requirement FW-3.5.6
3.7	E	Minor editorial change in justification part of requirement FW-3.7.5
3.7	C	Expanded the set of test case which verify requirement FW-3.7.5
4.1	C	Changed verification method of requirement FW-4.1.2 from 'A' to 'R'
4.3	E	Fixed typos in requirement FW-4.3.1
4.3	N	Added a note to requirement FW-4.3.2 to clarify the definition of "configuration" and modified requirement justification to reflect this definition
4.4	L	Fixed typos and improved formulation of requirement FW-4.4.2
4.5	L	Minor clarification in formulation of requirement FW-4.5.2

Section	Type	Description
5	N	New section on functional requirements of RT Containers
6.1	C	Changed verification method of requirement FW-6.1.1 from 'T' to 'R'
6.2	C	Changed verification method of requirements in this section from 'T' to 'R'
6.2	N	Added requirement FW-6.2.3 on RTD Internal Structure
6.2	E	Minor editorial changes to requirements FW-6.2.1 and FW-6.2.2
6.3	C	Extended requirement FW-6.3.1 on code memory footprint to also cover RT container; changed verification method for all requirements in this section from "Testing" to "Review"
6.3	N	Renumbered requirements FW-6.3.4 (State Machine Execution Time) and FW-6.3.5 (Procedure Execution Time) to, respectively, FW-6.3.5 and FW-6.3.6 and added a new requirement FW-6.3.4 cover memory footprint of RTDs
6.3	E	Fixed typo in references to user manual
6.4	C	Modified requirement FW-6.4.1 on concurrent environment to apply only to state machine and procedure parts of the C1 Implementation and changed its verification method from 'T' to 'R'
6.4	N	Added requirement FW-6.4.2 to cover concurrent use of RT containers
6.5	C	Modified formulation of requirement FW-6.5.1 on Test Coverage to refer to "system calls" rather than just "calls to `malloc` function"
6.5	N	New requirement FW-6.5.2 on stress testing of RT containers
6.6	C	Restricted requirement FW-6.6.1 on external libraries to state machine and procedure part of the C1 Implementation; changed verification method for this requirement from 'T' to 'R'
6.6	N	Added new requirement FW-6.6.2 on use of POSIX-compliant library
6.6	E	Removed erroenous reference to appendix E from requirement FW-6.6.1

Section	Type	Description
A	N	Added section A.3 on implementation of RT Container Concept
A.2	E	Fixed typo in introductory text of this section
B	E	In table 7: fixed typo in test case for `smNullCState` and in description of `smUndefinedTransSrc`; in table 8: changed `smUnreachablePNode` to `smUnreachableDNode`; in table 9: changed `FwSmTestCaseCheck4` to `FwPrTestCaseCheck4` and deleted entry for `FwSmTestCaseTransErr2` from bottom row of table
B	D	Deleted `FwPrTestCaseCheck3` from verification of `prTooManyActions`; deleted `FwPrTestCaseCheck7` from verification of `prIllNOfOutFlows`
C	N	Added section C.3 on verification Start/Stop behaviour of RT Containers
C.2	E	Fixed typos in introductory text of this section and in first column of table 12
D.1	E	Deleted duplicated entry for `FwSmTestCaseExecute3` from table 14
D.1	D	Deleted entry for `FwSmTestTrans3` from table 15
D.2	D	Deleted entry for `FwPrTestCaseExecute5` from first line of table 16
E	N	New appendix section on verification of notification behaviour of RT Containers

Table 4: Changes introduced in Revision 1.1.0

Section	Type	Description
3.2	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-3.2.2
3.5	E	Fixed typo in requirement FW-3.5.2
3.5	N	Added new requirement FW-3.5.3 on the order of evaluation of guards
3.7	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-3.7.3
4.2	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-4.2.3
4.5	N	Added new requirement FW-4.5.3 on the order of evaluation of guards
4.7	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-4.7.3
6.1	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-6.1.2
6.5	L	Replaced reference to Acceptance Test Procedure document with reference to User Manual in verification part of requirement FW-6.5.1
B	N	Added test cases to verify presence of unreachable states and unreachable pseudo-states in a SMD; Added test cases to verify presence of unreachable nodes in a PRD

2 Introduction

This document defines, justifies and verifies the user requirements for the *C1 Implementation*. The C1 Implementation is a C-language implementation of the State Machine Concept, of the Procedure Concept, and of the RT Container Concept of the *Framework (FW) Profile*. The FW Profile is a specification-level modelling language defined as a restriction of UML. The state machine concept of the FW Profile is modelled on the state machine concept of UML and the procedure concept of the FW Profile is modelled on the activity diagram concept of UML. The RT container concept is specific to the FW Profile. The FW Profile is defined in [1].

2.1 Intended Use of C1 Implementation

Although the C1 Implementation can be used wherever there is a need to implement a state machine or procedure concept or a threading model with a clear semantics, the high reliability of the implementation, the emphasis placed on formally specifying and verifying its expected behaviour, and the small demands on memory and processing resources mean that the C1 Implementation is especially well-suited for mission-critical embedded applications.

Thus, the intended use of the C1 Implementation is to support the implementation of the state machine, procedure, and RT container concepts of the FW Profile for mission-critical embedded applications.

2.2 Requirement Definition

Requirements are defined in tables with the following format:

FW-'x'/'V'	⟨**Requirement Title**⟩
Requirement	⟨Formulation of requirement⟩
Note	⟨Explicatory notes for requirement⟩
Justification	⟨Justification of requirement⟩
Implementation	⟨Dscription of how requirement is implemented⟩
Verification	⟨Dscription of how requirement is verified⟩

Here, the suffix 'x' is a numerical identifier which uniquely identifies the requirement within this document. The suffix 'V' identifies the verification method for the requirement according to the convention presented in section 2.2.3.

The explicatory notes are appended to the definition of the requirements where there is a need to clarify the terms which are used in their formulation.

In addition to their definition, this document also provides the following information for each requirement: a justification of the requirement; a description of how the requirement is implemented; and a description of how the requirement is verified.

2.2.1 Requirement Justification

For each requirement, a *justification* is provided which *validates* the requirement. Requirements are justified with respect to the intended use of the C1 Implementation. The intended use of the C1 Implementation is to support the implementation of the State Machine, Procedure and RT Container concepts of the FW Profile for mission-critical embedded applications (see section 2.1). Hence, a requirement is justified in proportion to its ability to further the adequacy of the C1 Implementation to support the implementation of the FW Profile in an environment where memory and processing resources are constrained and where reliability is of paramount importance.

2.2.2 Requirement Implementation

For each requirement, the function or data structure or other code-level construct in the source code which implements it is identified.

2.2.3 Requirement Verification

Verification information is provided for each requirement to demonstrate the correct implementation of the requirement. The following verification methods are possible:

- Verification by Review ('R'): the requirement is verified by inspecting the code or its documentation.
- Verification by Analysis ('A'): the requirement is verified by analysing the code, possibly with the help of a tool.
- Verification by Test ('T'): the requirement is verified by one or more test cases in the Test Suite.

One single verification method is defined for each requirement. This is identified as part of the requirement definition (see the description of the requirement format in section 2.2).

The Test Suite which is used for the verification by test is a complete application which demonstrates all aspects of the behaviour of the state machine, procedure, and RT container implementation. It consists of a sequence of Test Cases which are independent of each other. Each Test Case focuses on one particular functional aspect of the C1 Implementation. The Test Suite is distributed with the C1 Implementation. It is documented as part of the Doxygen documentation for the C1 Implementation and is described in the C1 Implementation User Manual (see reference [2]).

3 State Machine - Functional Requirements

This section defines the functional requirements for the State Machine part of the C1 Implementation. The functional requirements are those which define the functional behaviour of the state machines in the C1 Implementation.

FW-3.0.1/R	**Implementation of State Machine Concept**
REQUIREMENT	The C1 Implementation shall implement the state machine concept of the FW Profile of [1].
JUSTIFICATION	The intended use of the C1 Implementation is to support the implementation of the state machine concept of the FW Profile.
IMPLEMENTATION	The state machine behaviour specified by the FW Profile is implemented by the `FwSmCore.h` interface of the C1 Implementation.
VERIFICATION	The FW Profile defines state machines in terms of their *elements* and of their *behaviour*. The state machine behaviour is in turn defined in terms of three operations which can be performed upon a state machine (*start*, *stop*, and *command transition*). Appendix A.1 shows how each state machine element is mapped to a data structure in the C1 Implementation and how each state machine operation is mapped to a function in the `FwSmCore.h` interface of the C1 Implementation.

3.1 State Machine Descriptor (SMD) Requirements

FW-3.1.1/R	**State Machine Descriptor (SMD)**
REQUIREMENT	It shall be possible to address and to manipulate a state machine as a single entity (the *State Machine Descriptor* or SMD).
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The definition of the SMD simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The SMD is manipulated as an instance of type `FwSmDesc_t`.
VERIFICATION	A state machine is represented by an instance of type `struct FwSmDesc` and is manipulated as an instance of type `FwSmDesc_t` (a pointer to `struct FwSmDesc`). All functions which operate on a state machine take an instance of this type as their argument.

FW-3.1.2/R	**SMD Encapsulation**
REQUIREMENT	The SMD shall encapsulate all the information defining the configuration and the current state of its state machine.
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The encapsulation of all information related to a state machine in a single data structure simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The SMD is defined as an instance of type `struct FwSmDesc`.
VERIFICATION	The SMD models all the elements of a state machine (see section A.1) which define its configuration. It models the current state of a state machine in field `curState`.

3.2 Creation Requirements

FW-3.2.1/T	**Creation Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which a new SMD can be created.
JUSTIFICATION	Applications which wish to manipulate a state machine must first create the SMD which represents it.
IMPLEMENTATION	The state machine creation interface is implemented in `FwSmDCreate.h` and `FwSmSCreate.h`.
VERIFICATION	In the test suite, test state machines are used. The test state machines are created in functions with names like: `FwSmMakeTest<SM_Name>`. These make functions exercise all SMD creation functions offered by the C1 Implementation.

FW-3.2.2/T	**Dynamic and Static Creation**
REQUIREMENT	It shall be possible to create a new SMD either statically or dynamically.
NOTE	The term *static* refers to an instantiation process which does not rely on dynamic memory allocation.
JUSTIFICATION	Dynamic creation of state machine is convenient but may be forbidden in mission-critical applications.
IMPLEMENTATION	Dynamic state machine creation is implemented in `FwSmDCreate.h`. Static state machine creation is implemented in `FwSmSCreate.h`.
VERIFICATION	In the test suite, test state machines are used. By default, the test state machines are created dynamically in functions with names like: `FwSmMakeTest<SM_Name>`. Static creation of state machines is demonstrated in make functions with names like: `FwSmMakeTest<SM_Name>Static`.

FW-3.2.3/T	**Release of SMD**
REQUIREMENT	If an SMD is created dynamically, then it shall also be possible to destroy it dynamically by releasing the memory that was allocated to it.
JUSTIFICATION	The target applications for the C1 Implementation are mission-critical applications. In this domain, dynamic memory allocation is only allowed if the means are available to reclaim the dynamically allocated memory.

Implementation	Functions `FwSmRelease` and `FwSmReleaseRec` are provided in `FwSmDCreate.h` to release the memory allocated when a new SMD is created dynamically.
Verification	In most Test Cases in the Test Suite application, SMDs are created dynamically and the memory they use is then released before the end of the test. In the Acceptance Test for the C1 Implementation (see [2]), the Test Suite is run with the Valgrind tool and it is verified that no memory leaks occur and that all memory allocated dynamically is then released in an orderly way.

3.3 Configuration Requirements

FW-3.3.1/T	**Configuration Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which an SMD can be configured and made to match the characteristics of a certain state machine.
JUSTIFICATION	Applications which wish to manipulate a state machine must configure it before they can send transition commands to it.
IMPLEMENTATION	The state machine configuration interface is implemented in `FwSmConfig.h`.
VERIFICATION	In the test suite, test state machines are configured. The Test Suite exercises all configuration functions in `FwSmConfig.h`.

FW-3.3.2/T	**Reconfiguration of an SMD**
REQUIREMENT	It shall not be possible to re-configure an SMD which has already been configured.
NOTE	The term configuration refers to the operations which must be performed on a newly-created SMD before it can be started.
JUSTIFICATION	The C1 Implementation targets mission-critical applications. In this domain, dynamic reconfiguration of state machines would be regarded as unsafe because it makes it harder to determine behaviour through static analysis.

Implementation	The size of a state machine (the number of states, of choice pseudo-states, of transitions, of actions, and of guards) can only be set when its SMD is created and cannot therefore be changed after creation. After creation, three configuration operations must be performed on an SMD: (a) definition of taddhe states specified when the SMD was created (with function `FwSmAddState`); (b) definition of the choice pseudo-states specified when the SMD was created (with function `FwSmAddChoicePseudoState`); (c) definition of the transitions specified when the SMD was created (with functions with names like: `FwSmAddTrans*`). All of these operations add an item to an SMD (either a state, or a choice pseudo-state, or a transition). The SMD uses data structures with a fixed size. The configuration operations check whether there is space for the new item. If this is not the case, they return with an error. Since no functions are available for *removing* items from an SMD, it follows that, once an SMD has been configured, any attempt to execute a configuration function will fail with an error.
Verification	Test Case `FwSmTestCaseConfigErr2` verifies that attempts to reconfigure a state machine which has already been configured result in errors. More specifically, the test case creates and configures a state machine and then verifies that the following operations fail: (a) adding a new state; (b) re-defining an existing state; (c) adding a new choice pseudo-state; (d) re-defining an existing choice pseudo-state; (e) adding a new transition; (f) re-defining an existing transition.

FW-3.3.3/T	**Configuration Check**
Requirement	It shall be possible to check the completeness and correctness of the configuration of an SMD (*Configuration Check*).
Justification	The C1 Implementation targets mission-critical applications. In this domain, it is important to be able to periodically check the integrity of an application.

IMPLEMENTATION	The Configuration Check is implemented in functions `FwSmCheck` and `FwSmCheckRec`. These functions verify the completeness of the state machine configuration by checking that all states, choice pseudo-states, and transitions of a state machine have been defined. The correctness of the configuration of the state machine is checked indirectly as follows. The configuration functions in `FwSmConfig.h` perform a correctness check before executing a configuration request. If a violation of correctness is detected, it is reported by setting the *error code* field of the SMD. The Configuration Check verifies that the error code is set to "success". This value implies that no configuration errors have been detected and that therefore the configuration of the SMD is correct.
VERIFICATION	The ability of the Configuration Check to report an incomplete configuration is verified in the test cases: `FwSmTestCaseCheck1` to `FwSmTestCaseCheck6`. The test cases which verify the ability of the Configuration Check to detect incorrect configuration requests are identified in Appendix B.

FW-3.3.4/T	**Configuration Status Print**
REQUIREMENT	It shall be possible to extract and print the configuration information of an SMD.
JUSTIFICATION	This capability is useful during debugging.
IMPLEMENTATION	The configuration print service is implemented in function `FwSmPrintConfig`.
VERIFICATION	The configuration print service is verified in test cases `FwSmTestCasePrint1` and `FwSmTestCasePrint2`.

FW-3.3.5/R	**Configuration Constraints**
REQUIREMENT	The SMD configuration interface shall enforce the syntactical constraints C1 to C8 defined in section 4.2 of the FW Profile Definition in [1].
JUSTIFICATION	The C1 Implementation is intended to support the state machine concept of the FW Profile. Enforcement of these constraints is part of the support of the FW Profile model of a state machine.
IMPLEMENTATION	Enforcement of the constraints is achieved by the way the `FwSmAddTrans*` functions in `FwSmConfig.h` are defined: their interfaces make definition of a transition which violates a constraint impossible.

VERIFICATION

Constraint C1 ("The same pseudo-state cannot be both source and target for a transition") is enforced because the only `FwSmAddTrans*` functions which allow definition of a transition between two pseudo-states are `FwSmAddTransIpsToCps` and `FwSmAddTransCpsToFps` and these functions guarantee that source and destination are different.

Constraint C2 ("The source and target of a transition cannot both be choice pseudo-states") is enforced because there is no `FwSmAddTrans*` function which allows definition of a transition with a choice pseudo-state at both ends.

Constraint C3 ("The transition that has the initial pseudo-state as source can have neither a guard nor a trigger") is enforced because the transition out of the initial pseudo-state is defined through functions `FwSmAddTransIps*` and these functions allow neither a guard nor a trigger to be defined.

Constraint C4 has been deleted.

Constraint C5 ("Transitions that have a choice pseudo-state as source cannot have a transition trigger") is enforced because the transitions out of a choice pseudo-state are defined through functions `FwSmAddTransCps*` and these functions do not allow a trigger to be attached to the transition.

Constraint C6 has been deleted.

Constraint C7 ("Transitions that have a state as a source must have a transition command") is enforced because the transitions out of a state are defined through functions `FwSmAddTransSta*` and these functions require definition of a transition command.

Constraint C8 ("Transitions can only link states and/or pseudo-states that belong to the same state machine") is enforced because no functions are provided to let states or pseudo-states of different state machines be linked together through a transition.

3.4 Start and Stop Requirements

FW-3.4.1/T	**Start and Stop Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which the state machine represented by an SMD can be started, stopped and queried for its present Started/Stopped status.
JUSTIFICATION	The Start/Stop operations are defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the state machine concept of the FW Profile.
IMPLEMENTATION	The Start/Stop operations are implemented in `FwSmCore.h` by functions `FwSmStart` and `FwSmStop`. The status query operation is implemented by function `FwSmIsStarted`.
VERIFICATION	The Test Cases in the Test Suite start, stop and query state machines through the functions of `FwSmCore.h` and this interface has 100% coverage of its implementation.

FW-3.4.2/T	**Start Behaviour**
REQUIREMENT	The Start interface shall implement the behaviour defined in the activity diagram on the left-hand side of Figure 1.
JUSTIFICATION	The activity diagram of Figure 1 is the same as in the FW Profile definition of [1].
IMPLEMENTATION	The Start interface is implemented in function `FwSmStart`.
VERIFICATION	The verification of this requirement is done in Appendix C.1 where it is shown that all branches of the activity diagrams of 1 are covered by at least one Test Case in the Test Suite.

FW-3.4.3/T	**Stop Behaviour**
REQUIREMENT	The Stop interface shall implement the behaviour defined in the activity diagram on the right-hand side of Figure 1.
JUSTIFICATION	The activity diagram of Figure 1 is the same as in the FW Profile Definition Document of [1].
IMPLEMENTATION	The Stop interface is implemented in the function `FwSmStop`.

VERIFICATION	The verification of this requirement is done in Appendix C.1 where it is shown that all branches of the activity diagrams of 1 are covered by at least one Test Case in the Test Suite.

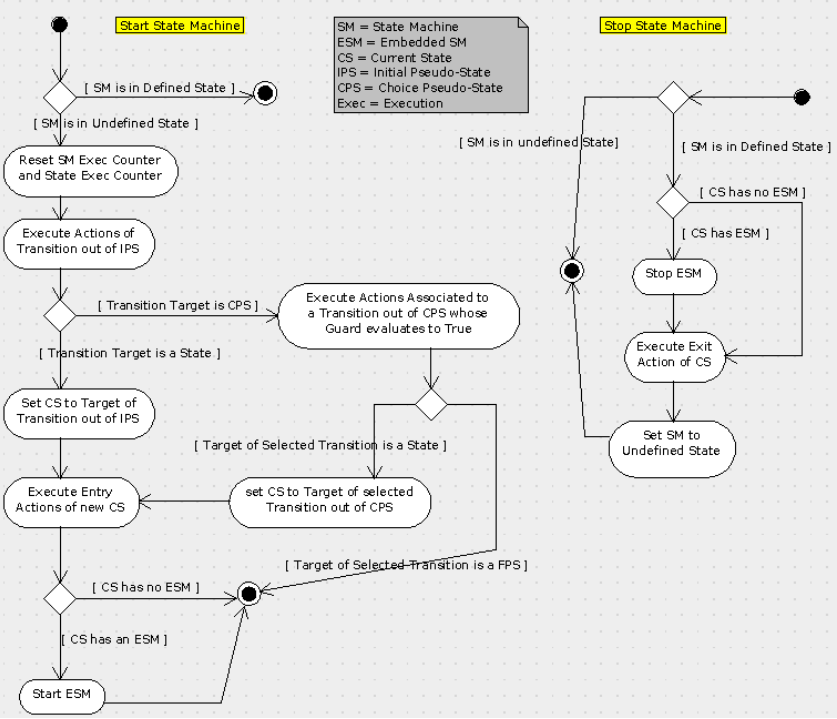

Fig. 1: State Machine Start/Stop Behaviour

3.5 Transition Command Requirements

FW-3.5.1/T	**Transition Command Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which Transition Commands can be sent to a state machine.
NOTE	A Transition Command is a command sent to a State Machine requesting it to perform a certain state transition.
JUSTIFICATION	The commanding of a state machine through transition commands is defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the state machine concept of the FW Profile.
IMPLEMENTATION	The Transition Command operations are implemented in `FwSmCore.h` by functions `FwSmMakeTrans` and `FwSmExecute`.
VERIFICATION	The Test Cases in the Test Suite send transition commands to state machines through the functions of `FwSmCore.h` and this interface has 100% coverage of its implementation.

FW-3.5.2/T	**Transition Command Behaviour**
REQUIREMENT	The C1 Implementation shall implement the Transition Command behaviour defined in the activity diagrams of Figures 2 and 3.
JUSTIFICATION	The activity diagrams of Figures 2 and 3 are the same as the activity diagrams in reference [1] which define the handling of transition commands in the FW Profile.
IMPLEMENTATION	The Transition Command behaviour is implemented in functions `FwSmMakeTrans` and (only for the "Execute" Transition Command) `FwSmExecute`.
VERIFICATION	The verification of this requirement is done in Appendix D.1 where it is shown that every branch of the activity diagrams of Figures 2 and 3 is covered by at least one Test Case in the Test Suite.

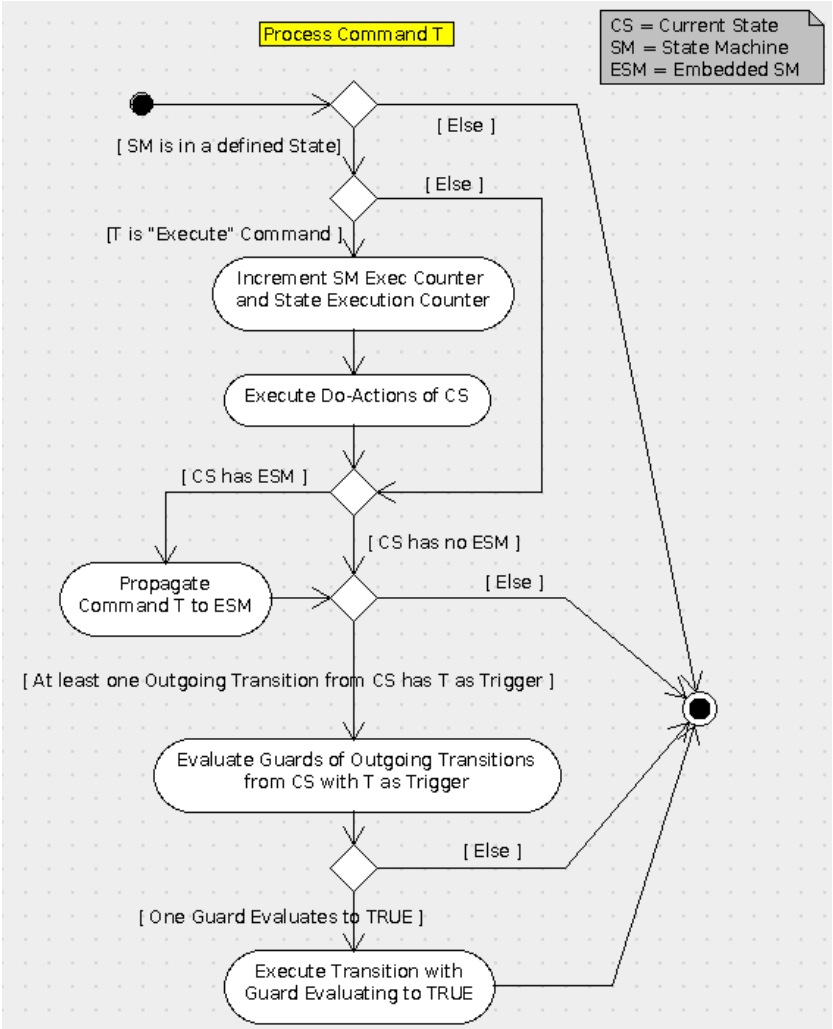

Fig. 2: Logic for Processing Transition Commands by a State Machine

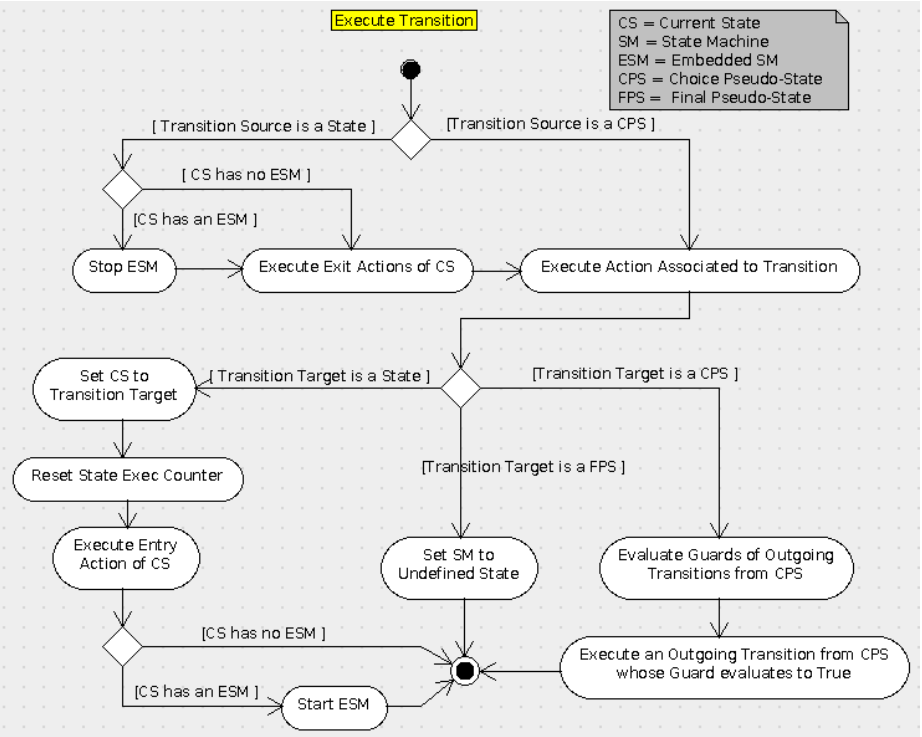

Fig. 3: Logic for Executing Transitions in a State Machine

FW-3.5.3/T	**Order of Evaluation of Guards**
REQUIREMENT	If processing of a transition command requires evaluation of several transition guards associated to different transitions out of the same state or choice pseudo-state, then the transition guards shall be evaluated in the order in which their transitions have been added to the state machine during the state machine configuration process.
NOTE	Transitions are "added" to a state machine during the state machine configuration process. This requirement assumes that they are added in sequence and one by one.
JUSTIFICATION	In principle, if the guards out of a state or choice pseudo-state are mutually exclusive and do not have side effects, their order of evaluation is irrelevant. For this reason, the FW Profile of [1] does not specify the order of evaluation of transition guards. However, in embedded applications with constrained CPU and memory resources, determinism in the order of evaluation may be exploited to improve run-time efficiency (for instance, by ensuring that the guard which is most likely to be true is evaluated first) and to implement "else" clauses in an efficient manner (see section 3.6 of [2]).

Implementation	The evaluation of the guards on the transitions out of a state is done in function `FwSmMakeTrans`. The evaluation of the guards on the transitions out of a choice pseudo-state is done in function `FwSmExecute`.
Verification	The order of evaluation is verified in test case `FwSmTestCaseTrans7` for transitions out of a state and in test case `FwSmTestCaseTrans8` for transitions out of a choice pseudo-state.

FW-3.5.4/A	**Commanding Interface**
Requirement	After having been completely and successfully configured, a state machine shall only ever change its internal state either in response to a Start/Stop command or in response to a Transition Command.
Justification	The state machine model of the FW Profile stipulates that the internal state of a state machine can only change in response to a Start/Stop command or in response to a Transition Command.
Implementation	The only functions in `FwSmCore.h` which can change the current state of a state machine are: `FwSmStart`, `FwSmStop`, `FwSmMakeTrans`, and `FwSmExecute`. These are precisely the functions which implement the Start/Stop behaviour and the response to a Transition Command.
Verification	A search for `curState` (the field of the SMD which holds the current state) in the `FwSmCore.c` file shows that this attribute is assigned to in only the following functions: `FwSmStart`, `FwSmStop`, `ExecTrans`. The last function is used by both `FwSmMakeTrans` and by `FwSmExecute`.

FW-3.5.5/T	**Dynamical Constraint**
Requirement	The C1 Implementation shall check compliance with the dynamical constraint D1 defined in section 5.2 of the FW Profile Definition in [1].
Justification	The D1 constraint represents a non-nominal situation which can only be recognized dynamically. The C1 Implementation targets mission critical applications where the ability to detect non-nominal situations is important to guarantee the integrity of a system.
Implementation	The check for the D1 condition is implemented in the `FwSmExecTrans` function. Its detection results in error `smTransErr`.

VERIFICATION	Test Cases `FwSmTestCaseTransErr1` and `FwSmTestCaseTransErr2` simulate situations where error `smTransErr` is reported because execution of a state transition encounters a choice pseudo-state which has no out-going transitions with a guard evaluating to true.

FW-3.5.6/T	**State Machine Data**
REQUIREMENT	As part of the processing of a Transition Command to a state machine, it shall be possible to exchange data with the actions and guards of the state machine.
JUSTIFICATION	The FW Profile stipulates that transition commands may carry data.
IMPLEMENTATION	Functions `FwSmMakeTrans` and `ExecTrans` in `FwSmCore.h` implement the execution of transition commands. Execution of transition commands is the only way to trigger the execution of the actions of a state machine or the evaluation of its guards. These two functions pass the SMD to the functions implementing the state machine actions and state machine guards. In the SMD, field `smData` is reserved to hold a pointer to a generic data structure. This data structure is intended to hold the data which are exchanged with the actions and guards of a state machine.
VERIFICATION	The Test State Machines used in the Test Cases use an instance of `struct TestSmData` as the means to exchange data with the actions and guards of a state machine. For example, the Test Cases `FwSmTestCaseExecute*` use this data structure to keep track of which actions are executed during a test.

FW-3.5.7/T	**State Machine State**
REQUIREMENT	It shall be possible to read the current state of a state machine.
JUSTIFICATION	Applications need to be able to check the current state of a state machine.
IMPLEMENTATION	Read-only access to the state of a state machine is provided by function `FwSmGetCurState`. Function `FwSmIsStarted` checks whether a state machine has been started.
VERIFICATION	Function `FwSmGetCurState` is used in virtually all Test Cases of the Test Suite. Function `FwSmIsStarted` is used in, for instance, Test Case `FwSmTestCaseStart1`.

3.6 Error Handling Requirements

FW-3.6.1/T	**Error Code**
Requirement	The SMD shall store the code of the last error encountered during the SMD configuration process or during the processing of transition commands.
Justification	The C1 Implementation targets embedded mission-critical applications where there normally is a need to periodically monitor the integrity of an application. The embedded character of the application, however, also means that memory resources are often limited and it may consequently not be possible to maintain a log of all errors. This requirement represents a compromise between these two needs in the sense that it allows an application to check whether an error has occured with only minimal storage requirements.
Implementation	The error code is stored in field `errCode` of the SMD.
Verification	The Test Cases with names like `FwSmTestCaseCheck*` and `FwSmTestCase*Err` test various error conditions and verify that the most recent error condition is correctly stored in the error code of an SMD. Appendix B lists the configuration and dynamic errors which may be reported in the error code of an SMD and, for each error, it identifies a Test Case where that error is reported and verified.

FW-3.6.2/T	**Access to Error Code**
Requirement	The SMD shall provide read-only access to the error code.
Justification	See the justification of the previous requirement.
Implementation	The value of the error code can be read with function `FwSmGetErrCode`.
Verification	See verification of the previous requirement.

3.7 Derived State Machine Creation Requirements

FW-3.7.1/T	**State Machine Extension Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which the SMD of a *derived state machine* can be created from the SMD of a *base state machine*.
JUSTIFICATION	The FW Profile defines an *adaptation mechanism* through which a new state machine can be built from an existing state machine by selectively modifying some of its elements. The extension mechanism of the C1 Implementation is an implementation of this adaptation mechanism.
IMPLEMENTATION	The state machine extension interface is implemented alongside the state machine creation interface in `FwSmDCreate.h` and `FwSmSCreate.h`.
VERIFICATION	In the test suite, derived state machines are used. The derived state machines are created in functions `FwSmMakeTestSMDer1` and `FwSmMakeTestSMDer1Static`. These make functions exercise all derived SMD creation functions offered by the C1 Implementation.

FW-3.7.2/T	**Dynamic and Static Creation of Derived SMD**
REQUIREMENT	It shall be possible to create the SMD of a new derived state machine either statically or dynamically.
NOTE	The term *static* refers to an instantiation process which does not rely on dynamic memory allocation.
JUSTIFICATION	Dynamic creation of a derived state machine is convenient, but may be forbidden in mission-critical applications.
IMPLEMENTATION	Dynamic creation of a derived state machine is implemented in `FwSmDCreate.h`. Static creation of a derived state machine creation is implemented in `FwSmSCreate.h`.
VERIFICATION	In the test suite, derived state machines are used. A derived state machine is created dynamically in function `FwSmMakeTestSMDer1` and statically in function `FwSmMakeTestSMDer1Static`.

FW-3.7.3/A	**Release of Derived SMD**
Requirement	If the SMD of a derived state machine is created dynamically, then it shall also be possible to destroy it dynamically by releasing the memory that was allocated to it.
Justification	The target applications for the C1 Implementation are mission-critical applications. In this domain, dynamic memory allocation is only allowed if the means are available to reclaim the dynamically allocated memory.
Implementation	Function `FwSmReleaseDer` is provided in `FwSmDCreate.h` to release the memory allocated when a new derived SMD is created dynamically.
Verification	Test Cases `FwSmTestCaseDer1`, `FwSmTestCaseDer3`, `FwSmTestCaseDerConfigErr1` and `FwSmTestCaseDerEmbed1` manipulate derived state machines whose SMD is created dynamically. The memory used by these SMDs is released before the end of the test. As part of the Acceptance Test Procedure for the C1 Implementation (see [2]) the Test Suite is run with the Valgrind tool to check that there are no memory leaks and that all memory allocated dynamically is then released in an orderly way.

FW-3.7.4/T	**Time of Derived SMD Creation**
Requirement	It shall be possible to create a derived state machine from a base state machine at any time after the base state machine has been fully and correctly configured.
Note	This requirement in particular implies that state machine extension can be done on state machines which have already been started.
Justification	This requirement complements the requirement allowing dynamic creation of derived state machines: the intention behind both requirements is to allow an application to create a new derived state machine at any time and under any circumstances.
Implementation	Function `FwSmCreateDer` which creates the SMD for a new derived state machine performs no check on the state of the base state machine (but, if the base state machine is not fully and correctly configured, the behaviour of the derived state machine is undefined).

VERIFICATION	Test Cases `FwSmTestCaseDer1`, `FwSmTestCaseDer3`, `FwSmTestCaseDerConfigErr1` and `FwSmTestCaseDerEmbed1` manipulate derived state machines whose SMD is created dynamically. Test Cases `FwSmTestCaseDer1`, `FwSmTestCaseDerConfigErr1` and `FwSmTestCaseDerEmbed1` derive the new SMD from a base state machine which has not yet been started whereas Test Case `FwSmTestCaseDer3` derives it from a base state machine which has already been started.

FW-3.7.5/T	**Configuration of Derived SMD at Creation**
REQUIREMENT	After successful creation, the SMD of a derived state machine shall be a structural clone of the SMD of the base state machine.
NOTE	The expression *structural clone* must be understood as follows. State machine B is a structural clone of state machine A if the following conditions are satisfied: A has the same states with the same actions as B; A has the same choice pseudo-states as B; A has the same transitions between the same states or choice pseudo-states and with the same actions and guards as B; if state S1 of A has an embedded state machine A1, then state S1 of B has an embedded state machine which is a structural clone of A1.
JUSTIFICATION	The state machine extension mechanism is intended to represent the state machine adaptation mechanism of the FW Profile. The adaptation mechanism of FW Profile is implemented in the C1 Implementation through a 2-step process: first, a clone of the base state machine is obtained by extending the base state machine and then selected elements of the derived state machine are overridden. This implementation of the adaptation mechanism therefore requires that a newly derived state machine be a clone of its base state machine.

IMPLEMENTATION	The SMD (i.e. the `struct FwSmDesc`) is internally split into two parts: the *extension descriptor* and the *base descriptor*. The base descriptor holds the information about the topology of a state machine (its states and choice pseudo-states and their inter-connections). The extension descriptor holds the information about the actions, guards, and embedded state machines. The base descriptor is shared between a base state machine and its derived state machines. Hence, a derived state machine is guaranteed by design to have the same topology as its base state machine. The equality of the guards and actions and the fact that embedded state machines of homologous states are structural clones is implemented in the `FwSmCreateDer` function (for the case of dynamic state machine extension) and in the `FwSmInitDer` function (for the case of static state machine extension).
VERIFICATION	Test Cases `FwSmTestCaseDerConfig1` and `FwSmTestCaseDerConfig2` verify that a derived state machine and its base state machine have the same actions and guards for the case of, respectively, dynamic and static state machine extension. Test cases `FwSmTestCaseDer3` and `FwSmTestCaseDer3` verify that a derived state machine and its base have the same behaviour for the case of, respectively, dynamic and static state machine extension.

FW-3.7.6/T	**State of Derived SMD at Creation**
REQUIREMENT	After successful creation, the SMD of a derived state machine shall be in the Stopped state.
JUSTIFICATION	This requirement enhances determinism of behaviour and determinism of behaviour is important in mission-critical applications.
IMPLEMENTATION	The initial state of a derived state machine is set in the `FwSmCreateDer` function (for the case of dynamic state machine extension) and in the `FwSmInitDer` function (for the case of static state machine extension).
VERIFICATION	Test Cases `FwSmTestCaseDerConfig1` and `FwSmTestCaseDerConfig2` verify that a derived state machine is in the Stopped state at creation for the case of, respectively, dynamic and static state machine extension.

FW-3.7.7/T	**Error Code of Derived SMD at Creation**

Requirement	After successful creation, the error code of a derived state machine shall be the same as the error code of the base state machine.
Justification	Extension of a state machine should only be done if the base state machine is correctly and fully configured (namely if no errors are reported by its error code). If this constraint is not satisfied, then the derived state machine cannot be assumed to be properly configured. The fact that its error code is the same as the (non-nominal) error code of its base state machine makes it easier for an application to detect a situation where a state machine has been derived from a base state machine which was not correctly configured (and which therefore had its error code set to a non-nominal value).
Implementation	The initial value of the error code of a derived state machine is set in the `FwSmCreateDer` function (for the case of dynamic state machine extension) and in the `FwSmInitDer` function (for the case of static state machine extension).
Verification	Test Cases `FwSmTestCaseDerConfig1` and `FwSmTestCaseDerConfig2` verify that a derived state machine has the same error code as its base state machine for the case of, respectively, dynamic and static state machine extension.

3.8 Derived State Machine Configuration Requirements

FW-3.8.1/T	**Action Override**
REQUIREMENT	After a derived state machine has been successfully created, it shall be possible to override one or more of its actions with a new action (*action override operation*).
JUSTIFICATION	The overriding of an action is one of the adaptation mechanisms mandated by the FW Profile.
IMPLEMENTATION	The action override operation is implemented by function `FwSmOverrideAction` in `FwSmConfig.h`.
VERIFICATION	The action override mechanism is used in the Test State Machine `SM1Der` created by function `FwSmMakeTestSMDer1` (dynamic creation) and `FwSmMakeTestSMDer1Static` (static creation). This Test State Machine is used in Test Cases `FwSmTestCaseDer2` and `FwSmTestCaseDer5`.

FW-3.8.2/R	**Overridden Action**
REQUIREMENT	The execution of the *action override operation* (see previous requirement) shall require knowledge of the identity of the overridden action.
NOTE	This requirement implies that it must not be possible to specify that a derived SMD overrides, say, the entry action of a certain state in the base state machine. This must only be possible if the name of the overridden action is known.
JUSTIFICATION	Ideally, it would be desirable to have a mechanism through which a base state machine can declare that certain actions are "final" and cannot therefore be overridden. Implementation of such a mechanism is judged too onerous in terms of memory and CPU requirements and is therefore regarded as unsuitable for an implementation aimed at embedded applications (which are often memory- and CPU-constrained). This requirement implies a more limited mechanism through which a base state machine can protect its actions from being overridden by keeping their identity private.
IMPLEMENTATION	Function `FwSmOverrideAction` requires as an argument the name of the function which implements the action to be overridden.

VERIFICATION	Function `FwSmOverrideAction` requires as an argument the name of the function which implements the actions to be overridden. Hence, an action can be overridden only if the name of the function implementing it is in scope. A base state machine can therefore prevent one of its actions from being overridden by keeping the function that implements it hidden (for instance, by declaring it as a `static` function).

FW-3.8.3/T	**Guard Override**
REQUIREMENT	After a derived state machine has been successfully created, it shall be possible to override one or more of its guards with a new guard (*guard override operation*).
JUSTIFICATION	The overriding of a guard is one of the adaptation mechanisms mandated by the FW Profile.
IMPLEMENTATION	The guard override operation is implemented by function `FwSmOverrideGuard` in `FwSmConfig.h`.
VERIFICATION	The guard override mechanism is used in the Test State Machine `SM1Der` created by function `FwSmMakeTestSMDer1` (dynamic creation) and `FwSmMakeTestSMDer1Static` (static creation). This Test State Machine is used in Test Cases `FwSmTestCaseDer2` and `FwSmTestCaseDer5`.

FW-3.8.4/T	**Overridden Guard**
REQUIREMENT	The execution of the *guard override operation* (see previous requirement) shall require knowledge of the identity of the overridden guard.
NOTE	This requirement implies that it must not be possible to specify that a derived SMD overrides, say, the guard of a certain state transition in the base state machine. This must only be possible if the name of the overridden guard is known.
JUSTIFICATION	Ideally, it would be desirable to have a mechanism through which a base state machine can declare that certain guards are "final" and cannot therefore be overridden. Implementation of such a mechanism is judged too onerous in terms of memory and CPU requirements and is therefore regarded as unsuitable for an implementation aimed at embedded applications (which are often memory- and CPU-constrained). This requirement implies a more limited mechanism through which a base state machine can protect its guards from being overridden by keeping their identity private.

Implementation	Function `FwSmOverrideGuard` requires as an argument the name of the function which implements the guard to be overridden.
Verification	Function `FwSmOverrideGuard` requires as an argument the name of the function which implements the guards to be overridden. Hence, a guard can be overridden only if the name of the function implementing it is in scope. A base state machine can therefore prevent one of its guards from being overridden by keeping the function that implements it hidden (for instance, by declaring it as a `static` function).

FW-3.8.5/T	**Embedding of State Machines**
Requirement	After a derived state machine has been successfully created, it shall be possible to embed a state machine within an "empty" state of the derived state machine.
Note	In the context of this requirement, a state is "empty" if it does not already hold an embedded state machine.
Justification	The embedding of a state machine into an "empty" state is one of the adaptation mechanisms mandated by the FW Profile.
Implementation	Function `FwSmEmbed` in `FwSmConfig.h` implements the state machine embedding mechanism. The function checks that the state within which the state machine must be embedded is empty and sets the error code if this is not the case.
Verification	Test Case `FwSmTestCaseDerEmbed1` verifies the embedding of a state machine with function `FwSmEmbed`. This Test Case also verifies that the embedding is only carried out if the target state is empty.

4 Procedure - Functional Requirements

This section defines the functional requirements of the Procedure part of the C1 Implementation. The functional requirements are those which define the functional behaviour of procedures in the C1 Implementation. It is recalled that the Procedures are loosely modelled on UML's Activity Diagrams.

FW-4.0.1/R	**Implementation of Procedure Concept**
Requirement	The C1 Implementation shall implement the procedure concept of the FW Profile of [1].
Justification	The intended use of the C1 Implementation is to support the implementation of the procedure concept of the FW Profile.
Implementation	The procedure behaviour specified by the FW Profile is implemented by the `FwPrCore.h` interface of the C1 Implementation.
Verification	The FW Profile defines procedures in terms of their *elements* and of their *behaviour*. The procedure behaviour is in turn defined in terms of three operations which can be performed upon a procedure. Appendix A.2 shows how each procedure element is mapped to a data structure in the C1 Implementation and how each procedure operation is mapped to a function in the `FwPrCore.h` interface of the C1 Implementation.

4.1 Procedure Descriptor (PRD) Requirements

FW-4.1.1/R	**Procedure Descriptor (PRD)**
REQUIREMENT	It shall be possible to address and to manipulate a procedure as a single entity (the *Procedure Descriptor* or PRD).
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The definition of the PRD simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The PRD is defined by type `FwPrDesc_t`.
VERIFICATION	A procedure is represented by an instance of type `struct FwPrDesc` and is manipulated as an instance of type `FwPrDesc_t`. All functions which operate on a procedure take an instance of this type as their argument.

FW-4.1.2/R	**PRD Encapsulation**
REQUIREMENT	The PRD shall encapsulate all the information defining the configuration and the current state of its procedure.
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The encapsulation of all information related to a procedure in a single data structure simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The PRD is defined as an instance of type `struct FwPrDesc`.
VERIFICATION	The PRD models all the elements of a procedure which define its configuration (see Appendix A.2). The state of a procedure is defined by its current node. The PRD models the current node of a procedure in field `curNode`.

4.2 Creation Requirements

FW-4.2.1/T	Creation Interface
Requirement	The C1 Implementation shall provide an interface through which a new PRD can be created.
Justification	Applications which wish to manipulate a procedure must first create the PRD which represents it.
Implementation	The procedure creation interface is implemented in `FwPrDCreate.h` and `FwPrSCreate.h`.
Verification	In the test suite, test procedures are used. The test procedures are created in functions with names like: `FwPrMakeTest<PR_Name>`. These make functions exercise all PRD creation functions offered by the C1 Implementation.

FW-4.2.2/T	Dynamic and Static Creation
Requirement	It shall be possible to create a new PRD either statically or dynamically.
Note	The term *static* refers to an instantiation process which does not rely on dynamic memory allocation.
Justification	Dynamic creation of procedure is convenient but may be forbidden in mission-critical applications.
Implementation	Dynamic procedure creation is provided by interface `FwPrDCreate.h`. Static procedure creation is provided by interface `FwPrSCreate.h`.
Verification	In the test suite, test procedures are used. By default, the test procedures are created dynamically in functions with names like: `FwPrMakeTest<PR_Name>`. Static creation of procedures is demonstrated in make functions with names like: `FwPrMakeTest<PR_Name>Static`.

FW-4.2.3/T	Release of PRD
Requirement	If a PRD is created dynamically, then it shall also be possible to destroy it dynamically by releasing the memory that was allocated to it.
Justification	The target applications for the C1 Implementation are mission-critical applications. In this domain, dynamic memory allocation is only allowed if the means are available to reclaim the dynamically allocated memory.

Implementation	Functions `FwPrRelease` and `FwPrReleaseRec` are provided in `FwPrDCreate.h` to release the memory allocated when a new PRD is created dynamically.
Verification	In most Test Cases in the Test Suite application, PRDs are created dynamically and the memory they use is then released before the end of the test. In the Acceptance Test for the C1 Implementation (see [2], the Test Suite is run with the Valgrind tool and it is verified that no memory leaks occur and that all memory allocated dynamically is then released in an orderly way.

4.3 Configuration Requirements

FW-4.3.1/T	Configuration Interface
Requirement	The C1 Implementation shall provide an interface through which a PRD can be configured and made to match the characteristics of a certain procedure.
Justification	Applications which wish to manipulate a procedure must configure it before they can send execution requests to it.
Implementation	The procedure configuration interface is implemented in `FwPrConfig.h`.
Verification	In the test suite, test procedures are configured. The Test Suite exercises all configuration functions in `FwPrConfig.h` (it has 100% statement coverage of the implementation of this interface).

FW-4.3.2/T	Reconfiguration of a PRD
Requirement	It shall not be possible to re-configure a PRD which has already been configured.
Note	The term *configuration* refers to the operations which must be performed on a newly created PRD before it can be started.
Justification	The C1 Implementation targets mission-critical applications. In this domain, dynamic reconfiguration of procedures would be regarded as unsafe because it makes it harder to determine behaviour through static analysis.

Implementation	The size of a procedure (the number of nodes and of control flows) can only be set when its PRD is created. It therefore cannot be modified after the PRD has been created. After creation, three configuration operations must be performed on a PRD: (a) definition of the action nodes specified when the PRD was created (with function `FwPrAddActionNode`); (b) definition of the decision nodes specified when the PRD was created (with function `FwPrAddDecisionNode`); (c) definition of the control flows specified when the PRD was created (with functions with names like: `FwPrAddFlow*`). All of these operations add an item to a PRD (either a node or a control flow). The PRD uses data structures with a fixed size. The configuration operations check whether there is space for the new item. If this is not the case, they return with an error. Since no functions are available for *removing* items from a PRD, it follows that, once the configuration of a PRD has been completed, any attempt to execute a configuration function will fail with an error.
Verification	Test Case `FwPrTestCaseCheck3` verifies that the following operations result in an error: (a) redefine an existing action node; (b) redefine an existing decision node; (c) add a new action node to a PRD which is already configured; (d) add a new decision node to a PRD which is already configured. Test Case `FwPrTestCaseCheck5` verifies that attempts to redefine a control flow or to add a new control flow to a procedure which is already configured result in an error.

FW-4.3.3/T	**Configuration Check**
Requirement	It shall be possible to check the completeness and correctness of the configuration of a PRD (*Configuration Check*).
Justification	The C1 Implementation targets mission-critical applications. In this domain, it is important to be able to periodically check the integrity of an application. The configuration check serves this purpose.

IMPLEMENTATION	The Configuration Check is implemented in function `FwPrCheck`. This function verifies the completeness of the procedure configuration by checking that all nodes and control flows of the procedure have been defined. The correctness of the configuration of the procedure is checked indirectly as follows. The configuration functions in `FwPrConfig.h` perform a correctness check before executing a configuration request. If a violation of correctness is detected, it is reported by setting the *error code* field of the PRD. The Configuration Check verifies that the error code is set to "success". This value implies that no configuration errors have been detected and that therefore the configuration of the PRD is correct.
VERIFICATION	The ability of the Configuration Check to report an incomplete configuration is verified in the test cases: `FwPrTestCaseCheck1` to `FwPrTestCaseCheck7`. The test cases which verify the ability of the Configuration Check to detect incorrect configuration requests are identified in appendix B.

4.4 Start and Stop Requirements

FW-4.4.1/T	**Start and Stop Interface**
Requirement	The C1 Implementation shall provide an interface through which the procedure represented by a PRD can be started, stopped and queried for its current Started/Stoped status.
Justification	The Start/Stop operations are defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the procedure concept of the FW Profile.
Implementation	The Start/Stop operations are implemented in `FwPrCore.h` by functions `FwPrStart` and `FwPrStop`. The status query operation is implemented by function `FwPrIsStarted`.
Verification	The Test Cases in the Test Suite start and stop procedures through the functions of the `FwPrCore.h` module and this module has 100% coverage.

FW-4.4.2/T	**Start and Stop Behaviour**
Requirement	The Start and Stop interface shall implement the behaviour defined in the activity diagram of Figure 4.
Justification	The activity diagram of Figure 4 is the same as in the FW Profile definition of [1].
Implementation	The Start interface is implemented by function `FwPrStart`. The Stop interface is implemented by function `FwPrStop`.
Verification	The verification of this requirement is done in section C.2.

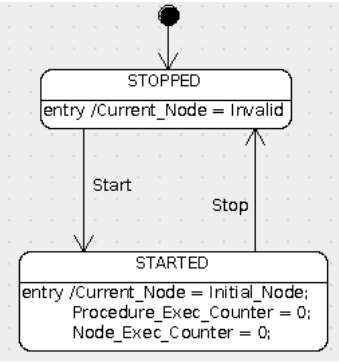

Fig. 4: Procedure Start/Stop Behaviour

4.5 Execution Requirements

FW-4.5.1/T	**Execution Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which a procedure can be executed.
NOTE	A procedure is executed by sending an "Execute" command to it.
JUSTIFICATION	The executability of a procedure is defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the procedure concept of the FW Profile.
IMPLEMENTATION	The execution interface is implemented in `FwPrCore.h` by function `FwPrExecute`.
VERIFICATION	The Test Cases in the Test Suite send execution commands to procedures through the functions of the `FwPrCore.h` module and this module has 100% coverage.

FW-4.5.2/T	**Execution Behaviour**
REQUIREMENT	The C1 Implementation shall implement the execution behaviour defined in the activity diagram of Figure 5.
JUSTIFICATION	The activity diagram of Figure 5 is the same as the activity diagram which defines the execution behaviour in the FW Profile definition of [1].
IMPLEMENTATION	The execution behaviour is implemented in function `FwPrExecute`.
VERIFICATION	The verification of this requirement is done in appendix D.2 where it is shown that every branch of the activity diagram of Figure 5 is covered by at least one Test Case in the Test Suite.

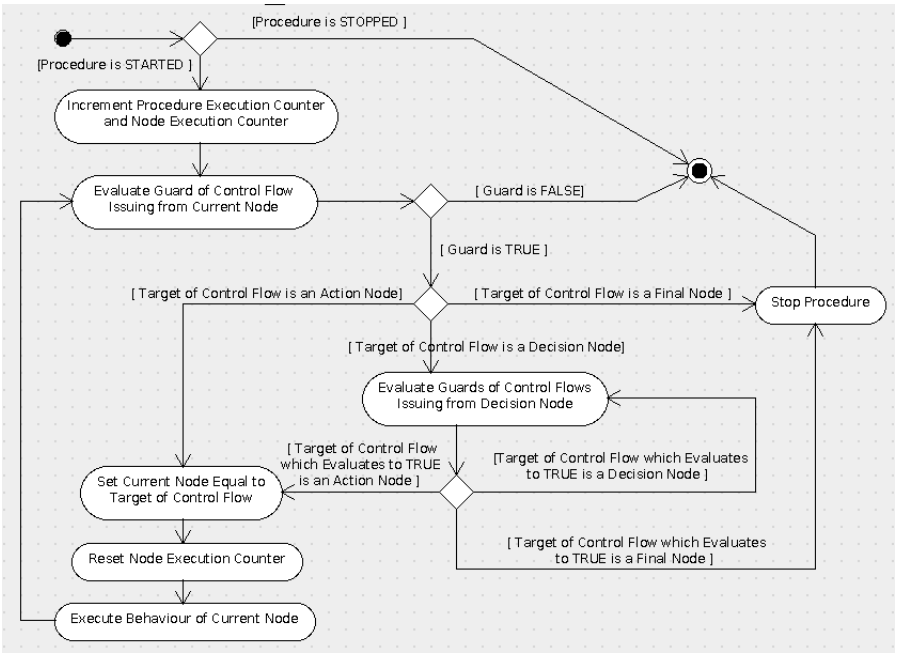

Fig. 5: Procedure Execution Logic

FW-4.5.3/T	**Order of Evaluation of Guards**
REQUIREMENT	If processing of an execution request requires evaluation of several control flow guards associated to different control flows out of the same decision node, then the control flow guards shall be evaluated in the order in which their control flows have been added to the procedure during the procedure configuration process.
NOTE	Control flows are "added" to a procedure during the procedure configuration process. This requirement assumes that they are added in sequence and one by one.
JUSTIFICATION	In principle, if the guards on the control flows out of a decision node are mutually exclusive and do not have side effects, their order of evaluation is irrelevant. For this reason, the FW Profile of [1] does not specify the order of evaluation of control flow guards. However, in embedded applications with constrained CPU and memory resources, determinism in the order of evaluation may be exploited to improve run-time efficiency (for instance, by ensuring that the guard which is most likely to be true is evaluated first) and to implement "else" clauses in an efficient manner (see section 4.6 of [2]).
IMPLEMENTATION	The evaluation of the guards on the control flows out of a decision node is done in function `FwPrExecute`.

VERIFICATION	The order of evaluation of control flow guards is verified in test case `FwPrTestCaseExecute9`.

FW-4.5.4/T	**Change of Internal State**
REQUIREMENT	After having been completely and successfully configured, a procedure shall only ever change its internal state either in response to a Start/Stop command or in response to an Execute Command.
JUSTIFICATION	The procedure model of the FW Profile stipulates that the internal state of a procedure can only change in response to a Start/Stop command or in response to an Execute Command.
IMPLEMENTATION	The current state of a procedure is the node at which the procedure is waiting. The only functions in `FwPrCore.h` which can change the current state of a procedure are: `FwPrStart`, `FwPrStop`, and `FwPrExecute`. These are precisely the functions which implement the Start/Stop behaviour and the response to a Transition Command.
VERIFICATION	A search for `curNode` (the field of the PRD which holds the current node) in the `FwPrCore.c` file shows that this attribute is assigned to in only the following functions: `FwPrStart`, `FwPrStop`, and `FwPrExecute`.

FW-4.5.5/T	**Dynamical Constraint**
REQUIREMENT	The C1 Implementation shall check compliance with the dynamical constraint D1 defined in section 4.2 of the FW Profile Definition in [1].
JUSTIFICATION	The D1 constraint represents a non-nominal situation which cannot be detected statically. The C1 Implementation targets mission critical applications where the ability to detect non- nominal situations is important to guarantee the integrity of a system.
IMPLEMENTATION	The check for the D1 condition is implemented in the `FwPrExecute` function. Its detection results in error `prFlowErr`.
VERIFICATION	Test Case `FwPrTestCaseCheck4` simulates a situation where error `prFlowErr` is reported because execution of a transition through a decision node finds that all out-going control flows have guards which evaluate to false.

FW-4.5.6/T	**Procedure Data**

REQUIREMENT	As part of the processing of an execution request to a procedure, it shall be possible to exchange data with the actions and guards of the procedure.
JUSTIFICATION	The FW Profile stipulates that execution commands may carry data.
IMPLEMENTATION	Function `FwPrExecute` in `FwPrCore.h` implements the code through which the actions of a procedure are executed and its guards are evaluated. These functions pass the PRD to the functions implementing the procedure actions and procedure guards. In the PRD, field `prData` is reserved to hold a pointer to a generic data structure. This data structure is intended to hold the data which are exchanged with the actions and guards of a procedure.
VERIFICATION	The Test Procedures used in the Test Cases use an instance of `struct TestPrData` as the means to exchange data with the actions and guards of a procedures. For example, the Test Cases `FwPrTestCaseExecute*` use this data structure to keep track of which actions are executed during a test.

FW-4.5.7/T	**Procedure State**
REQUIREMENT	It shall be possible to read the current state of a procedure.
JUSTIFICATION	Applications need to be able to check the state of a procedure.
IMPLEMENTATION	The state of a procedure is determined by its Stopped/Started state and by its current node. Read-only access to the current node of a procedure is provided by function `FwPrGetCurNode`. Function `FwPrIsStarted` checks whether a procedure has been started.
VERIFICATION	Function `FwPrGetCurNode` is used in virtually all Test Cases of the Test Suite. Function `FwPrIsStarted` is used in, for instance, Test Case `FwPrTestCaseExecute8`.

4.6 Error Handling Requirements

FW-4.6.1/T	**Error Code**
Requirement	The PRD shall store the code of the last error encountered during the PRD configuration process or during the processing of execution requests.
Justification	The C1 Implementation targets embedded mission-critical applications where there normally is a need to periodically monitor the integrity of an application. The embedded character of the application, however, also means that memory resources are often limited and it may consequently not be possible to maintain a log of all errors. This requirement represents a compromise between these two needs in the sense that it allows an application to check whether an error has occured with only minimal storage requirements.
Implementation	The error code is stored in field `errCode` of the PRD.
Verification	The Test Cases with names like `FwPrTestCaseCheck*` test various error conditions and verify that the most recent error condition is correctly stored in the error code of an PRD. Appendix B lists the configuration and dynamic errors which may be reported in the error code of a PRD and, for each error, it identifies a Test Case where that error is reported and verified.

FW-4.6.2/T	**Access to Error Code**
Requirement	The PRD shall provide read-only access to the error code.
Justification	See the justification of the previous requirement.
Implementation	The value of the error code can be read with function `FwPrGetErrCode`.
Verification	See verification of the previous requirement.

4.7 Derived Procedure Creation Requirements

FW-4.7.1/T	**Extension Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which the PRD of a *derived procedure* can be created from the PRD of a *base procedure*.
JUSTIFICATION	The FW Profile defines an *adaptation mechanism* through which a new procedure can be built from an existing procedure by selectively modifying some of its elements. The extension mechanism of the C1 Implementation is an implementation of this adaptation mechanism.
IMPLEMENTATION	The procedure extension interface is implemented alongside the procedure creation interface in modules `FwPrDCreate.h` and `FwPrSCreate.h`.
VERIFICATION	In the test suite, derived procedures are used. The derived procedures are created in functions `FwPrMakeTestPRDer1` and `FwPrMakeTestPRDer1Static`. These make functions exercise all derived PRD creation functions offered by the C1 Implementation.

FW-4.7.2/T	**Dynamic and Static Derived PRD Creation**
REQUIREMENT	It shall be possible to create the PRD of a derived procedure either statically or dynamically.
NOTE	The term *static* refers to an instantiation process which does not rely on dynamic memory allocation.
JUSTIFICATION	Dynamic creation of a derived procedure is convenient, but may be forbidden in mission-critical applications.
IMPLEMENTATION	Dynamic creation of a derived procedure is defined in `FwPrDCreate.h`. Static creation of a derived procedure creation is defined in `FwPrSCreate.h`.
VERIFICATION	In the Test Suite, derived procedures are used. A derived procedure is created dynamically in function `FwPrMakeTestPRDer1` and statically in function `FwPrMakeTestPRDer1Static`.

FW-4.7.3/T	**Release of Derived PRD**
REQUIREMENT	If the PRD of a derived procedure is created dynamically, then it shall also be possible to destroy it dynamically by releasing the memory that was allocated to it.

JUSTIFICATION	The target applications for the C1 Implementation are mission-critical applications. In this domain, dynamic memory allocation is only allowed if the means are available to reclaim the dynamically allocated memory.
IMPLEMENTATION	Function `FwPrReleaseDer` is provided in `FwPrDCreate.h` to release the memory allocated when a new derived PRD is created dynamically.
VERIFICATION	Test Cases `FwPrTestCaseDer1` and `FwPrTestCaseDer2`, manipulate derived procedures whose PRD is created dynamically. The memory used by these PRDs is released before the end of the test. In the Acceptance Test for the C1 Implementation (see [2], the Test Suite is run with the Valgrind tool and it is verified that no memory leaks occur and that all memory allocated dynamically is then released in an orderly way.

FW-4.7.4/T	**Time of Derived PRD Creation**
REQUIREMENT	It shall be possible to create a derived procedure from a base procedure at any time after the base procedure has been fully and correctly configured.
NOTE	This requirement in particular implies that procedure extension can be done on procedures which have already been started.
JUSTIFICATION	This requirement complements the requirement allowing dynamic creation of derived procedures: the intention behind both requirements is to allow an application to create a new derived procedure at any time and under any circumstances.
IMPLEMENTATION	Function `FwPrCreateDer` which creates the PRD for a new derived procedure performs no check on the state of the base procedure (but, if the base procedure is not fully and correctly configured, the behaviour of the derived procedure is undefined).
VERIFICATION	Test Cases `FwPrTestCaseDer1` and `FwPrTestCaseDer2` manipulate derived procedures whose PRD is created dynamically. Test Case `FwPrTestCaseDer1` derives the new PRD from a base procedure which has not yet been started whereas Test Case `FwPrTestCaseDer2` derives it from a base procedure which has already been started and executed.

FW-4.7.5/T	**Configuration of Derived PRD at Creation**

Requirement	After successful creation, the PRD of a derived procedure shall be a structural clone of the PRD of the base procedure.
Note	The expression *structural clone* must be understood as follows. Procedure B is a structural clone of procedure A if the following conditions are satisfied: A has the same action nodes with the same actions as B; A has the decision nodes as B; A has the same control flows between the same nodes and with the same guards as B.
Justification	The procedure extension mechanism is intended to represent the procedure adaptation mechanism of the FW Profile. The adaptation mechanism of FW Profile is implemented in the C1 Implementation through a 2-step process: first, a clone of the base procedure is obtained by extending the base procedure and then selected elements of the derived procedure are overridden. This implementation of the adaptation mechanism therefore requires that a newly derived procedure be a clone for all its structural elements of the base procedure.
Implementation	The PRD (i.e. the `struct FwPrDesc`) is internally split into two parts: the *extension descriptor* and the *base descriptor*. The base descriptor holds the information about the topology of a procedure (its action nodes and decision nodes and their inter-connections). The extension descriptor holds the information about the actions and guards. The base descriptor is shared between a base procedure and its derived procedures. Hence, a derived procedure is guaranteed by design to have the same topology as its base procedure. The equality of the guards and actions is implemented in the `FwPrCreateDer` function (for the case of dynamic procedure extension) and in the `FwPrInitDer` function (for the case of static procedure extension).
Verification	Test Case `FwPrTestCaseDerCheck1` verifies that a derived procedure and its base procedure are structural clones of each other both for the case of dynamic and for the case of static procedure extension.

FW-4.7.6/T	**State of Derived PRD at Creatio**
Requirement	After successful creation, the PRD of a derived procedure shall be in the Stopped state.
Justification	This requirement enhances determinism of behaviour and determinism of behaviour is important in mission-critical applications.

IMPLEMENTATION	The initial state of a derived procedure is set in the `FwPrCreateDer` function (for the case of dynamic procedure extension) and in the `FwPrInitDer` function (for the case of static procedure extension).
VERIFICATION	Test Case `FwPrTestCaseDerCheck3` verifies that a dynamically derived procedure is in the Stopped state at creation. Test Case `FwPrTestCaseDerCheck5` does the same for a statically derived procedure.

FW-4.7.7/T	**Error Code of Derived PRD at Creation**
REQUIREMENT	After successful creation, the error code of a derived procedure shall be the same as the error code of the base procedure.
JUSTIFICATION	Extension of a procedure should only be done if the base procedure is correctly and fully configured (namely if no errors are reported by its error code). If this constraint is not satisfied, then the derived procedure cannot be assumed to be properly configured. The fact that its error code is the same as the (non-nominal) error code of its base procedure makes it easier for an application to detect the problem.
IMPLEMENTATION	The initial value of the error code of a derived procedure is set in the `FwPrCreateDer` function (for the case of dynamic procedure extension) and in the `FwPrInitDer` function (for the case of static procedure extension).
VERIFICATION	Test Case `FwPrTestCaseDerCheck3` verifies that a dynamically derived procedure has the same error code as its base procedure. Test Case `FwPrTestCaseDerCheck5` does the same for a statically derived procedure.

4.8 Derived Procedure Configuration Requirements

FW-4.8.1/T	**Action Override**
REQUIREMENT	After a derived procedure has been successfully created, it shall be possible to override one or more of its actions with a new action (*action override operation*).
JUSTIFICATION	The overriding of an action is one of the adaptation mechanisms mandated by the FW Profile.
IMPLEMENTATION	The action override operation is implemented by function `FwPrOverrideAction` in `FwPrConfig.h`.
VERIFICATION	The action override mechanism is used in the Test Procedure `PR1Der` created by function `FwPrMakeTestPRDer1` (dynamic creation) and `FwPrMakeTestPRDer1Static` (static creation). This Test Procedure is used in Test Cases `FwPrTestCaseDer2` (dynamic creation) and `FwPrTestCaseDer3` (static creation).

FW-4.8.2/T	**Overridden Action**
REQUIREMENT	The execution of the *action override operation* shall require knowledge of the identity of the overridden action.
NOTE	This requirement implies that it must not be possible to specify that a derived PRD overrides the action of a certain node in the base procedure. This must only be possible if the name of the overridden action is known.
JUSTIFICATION	Ideally, it would be desirable to have a mechanism through which a base procedure can declare that certain actions are "final" and cannot therefore be overridden. Implementation of such a mechanism is judged too onerous in terms of memory and CPU requirements and is therefore regarded as unsuitable for an implementation aimed at embedded applications (which are often memory- and CPU-constrained). This requirement implies a more limited mechanism through which a base procedure can protect its actions from being overridden by keeping their identity private.
IMPLEMENTATION	Function `FwPrOverrideAction` requires as an argument the name of the function which implements the actions to be overridden.

VERIFICATION	Function `FwPrOverrideAction` requires as an argument the name of the function which implements the action to be overridden. Hence, an action can be overridden only if the name of the function implementing it is in scope. A base procedure can therefore prevent one of its actions from being overridden by keeping the function that implements it hidden (for instance, by declaring it as a `static` function).

FW-4.8.3/T	**Guard Override**
REQUIREMENT	After a derived procedure has been successfully created, it shall be possible to override one or more of its guards with a new guard (*guard override operation*).
JUSTIFICATION	The overriding of a guard is one of the adaptation mechanisms mandated by the FW Profile.
IMPLEMENTATION	The guard override operation is implemented by function `FwPrOverrideGuard` in `FwPrConfig.h`.
VERIFICATION	The guard override mechanism is used in the Test Procedure `PR1Der` created by function `FwPrMakeTestPRDer1` (dynamic creation) and `FwPrMakeTestPRDer1Static` (static creation). This Test Procedure is used in Test Cases `FwPrTestCaseDer2` (dynamic creation) and `FwPrTestCaseDer3` (static creation).

FW-4.8.4/T	**Overridden Guard**
REQUIREMENT	The execution of the *guard override operation* shall require knowledge of the identity of the overridden guard.
NOTE	This requirement implies that it must not be possible to specify that a derived PRD overrides, say, the guard of a certain control flow in the base procedure. This must only be possible if the name of the overridden guard is known.
JUSTIFICATION	Ideally, it would be desirable to have a mechanism through which a base procedure can declare that certain guards are "final" and cannot therefore be overridden. Implementation of such a mechanism is judged too onerous in terms of memory and CPU requirements and is therefore regarded as unsuitable for an implementation aimed at embedded applications (which are often memory- and CPU-constrained). This requirement implies a more limited mechanism through which a base procedure can protect its guards from being overridden by keeping their identity private.

Implementation	Function `FwPrOverrideGuard` requires as an argument the name of the function which implements the guards to be overridden.
Verification	Function `FwPrOverrideGuard` requires as an argument the name of the function which implements the guards to be overridden. Hence, a guard can be overridden only if the name of the function implementing it is in scope. A base procedure can therefore prevent one of its guards from being overridden by keeping the function that implements it hidden (for instance, by declaring it as a `static` function).

5 RT Containers - Functional Requirements

This section defines the functional requirements for the RT Container part of the C1 Implementation. The functional requirements are those which define the functional behaviour of the RT containers in the C1 Implementation.

FW-5.0.1/R	**Implementation of RT Container Concept**
REQUIREMENT	The C1 Implementation shall implement the RT container concept of the FW Profile of [1].
JUSTIFICATION	The intended use of the C1 Implementation is to support the implementation of the RT container concept of the FW Profile.
IMPLEMENTATION	The RT container behaviour specified by the FW Profile is implemented by the `FwRtCore.h` interface of the C1 Implementation.
VERIFICATION	The FW Profile defines RT containers in terms of their *elements* and of their *behaviour*. The container behaviour is in turn defined in terms of the behaviour of two procedures, of one thread, and of the three operations which can be performed upon a container (`Start`, `Stop`, and `Notify`). Appendix A.3 shows how the container elements and the container operations are mapped to data structures and functions in the C1 Implementation

5.1 RT Container Descriptor (RTD) Requirements

FW-5.1.1/R	**RT Container Descriptor (RTD)**
REQUIREMENT	It shall be possible to address and to manipulate a RT Container as a single entity (the *RT Container Descriptor* or RTD).
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The definition of the RTD simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The RTD is defined by type `struct FwRtDesc`.
VERIFICATION	A RT container is represented by an instance of type `struct FwRtDesc` and is manipulated as an instance of type `FwRtDesc_t`. All functions which operate on a RT container take an instance of this type as their argument.

FW-5.1.2/R	**RTD Encapsulation**
REQUIREMENT	The RTD shall encapsulate all the information defining the configuration and the current state of its container.
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The encapsulation of all information related to a RT container in a single data structure simplifies the interface between the modules of the C1 Implementation and the user application.
IMPLEMENTATION	The RTD is defined as an instance of type `struct FwRtDesc`.
VERIFICATION	The configuration information for a container is defined through the operations of `FwRtConfig.h`. These operation only manipulate the RTD. The state of a container is defined by the value of its Notification Counter and by the container state. These are mapped to fields `notifCounter` and `state` in the RTD.

5.2 Creation Requirements

FW-5.2.1/T	**Creation Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which a new RTD can be created.
JUSTIFICATION	Applications which wish to manipulate an RT container must first create the RTD which represents it.
IMPLEMENTATION	The RTD creation interface is trivial in the sense that RTDs are created by instantiating a variable of type `struct FwRtDesc`.
VERIFICATION	In every RT container test case in the test suite, one or more RTDs are created.

FW-5.2.2/T	**Dynamic and Static Creation**
REQUIREMENT	It shall be possible to create a new RTD either statically or dynamically.
NOTE	The term *static* refers to an instantiation process which does not rely on dynamic memory allocation.
JUSTIFICATION	Dynamic creation of containers is convenient but may be forbidden in mission-critical applications.
IMPLEMENTATION	RTDs are created by instantiating a variable of type `struct FwRtDesc`. Applications are free to instantiate this variable either statically (on the stack or as a global variable) or dynamically (through a call to `malloc`).
VERIFICATION	In every RT container test case in the test suite, one or more RTDs are created. In test case `FwRtTestCaseRunDefault1`, the RTD is created dynamically (using `malloc`); in all other test cases, the RTDs are created statically.

5.3 Configuration Requirements

FW-5.3.1/T	**Configuration Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which an RTD can be configured and made to match the characteristics of a certain RT Container.
JUSTIFICATION	Applications which wish to manipulate a RT Container must configure it before they can send notification requests to it.
IMPLEMENTATION	The container configuration interface is implemented in `FwRtConfig.h`.
VERIFICATION	In the test suite, test RT containers are configured. The Test Suite exercises all configuration functions in `FwRtConfig.h` (it has 100% statement coverage of the implementation of this interface with the exception of code which is entered when a POSIX system call fails).

FW-5.3.2/T	**Reconfiguration of an RTD**
REQUIREMENT	It shall not be possible to re-configure an RTD dynamically while it is being used to process notification requests.
NOTE	The term *configuration* refers to the operations which must be performed on a newly-created RTD before it can be started.
JUSTIFICATION	The C1 Implementation targets mission-critical applications. In this domain, dynamic reconfiguration of RT containers would be regarded as unsafe because it makes it harder to determine behaviour through static analysis.
IMPLEMENTATION	The RTD configuration functions (with the exception of the `FwRtReset` function and of the `FwRtSetData` function) check the container state and are only effective if the container is in state `rtContUninitialized`. In all other cases (i.e. during the container's operational use), they cause an error state to be entered. No such check is possible for the `FwRtReset` function because the purpose of this function is precisely to initialize the RTD and therefore no assumption can be made about the RTD state at the time the function is called. No such check is needed for the `FwRtSetData` function because the purpose of this function is to load the container data and this is an optional operation.

VERIFICATION	Test Case `FwRtTestCaseSetAction1` verifies that attempts to load a new container procedure action or to re-initialize the container during the container's operational phase lead to an error. Test Case `FwRtTestCaseSetAttr1` verifies that attempts to load a new POSIX attribute obect during the container's operational phase lead to an error.

FW-5.3.3/T	**Setting of POSIX Attributes**
REQUIREMENT	During the configuration process, it shall be possible to set the attributes of any POSIX object used by a RT container (see requirement FW-6.6.2).
JUSTIFICATION	The C1 Implementation targets mission-critical applications. In this domain, it is often important to fine-tune the real-time behaviour of threads. This requires access to the attributes to the POSIX objects upon which the real-time behaviour of a RT container is built.
IMPLEMENTATION	Function `FwRtSetMutexAttr` allows the POSIX attribute objects of a container's thread, of its mutex and of its condition variable to be set.
VERIFICATION	Test Case `FwRtTestCaseRunNonNullAttr1` verifies that it is possible to load non-NULL attributes for the POSIX thread, the POSIX mutex and the POSIX condition variable associated to a RT container.

5.4 Start and Stop Requirements

FW-5.4.1/T	**Start and Stop Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which the RT Container represented by an RTD can be started, stopped and queried for its current Started/Stopped status.
JUSTIFICATION	The Start/Stop operations are defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the RT container concept of the FW Profile.
IMPLEMENTATION	The Start/Stop operations are implemented in module `FwRtCore.h` by functions `FwRtStart` and `FwRtStop`. The status query operation is implemented by function `FwRtGetContState` which returns the current state of a container.
VERIFICATION	The Test Cases in the Test Suite start and stop RT containers and query them for their status through the functions of the `FwRtCore.h` module and this module has 100% coverage (except for branches entered in case of POSIX system call errors).

FW-5.4.2/T	**Start and Stop Behaviour**
REQUIREMENT	The Start and Stop interface shall implement the behaviour defined in the activity diagrams of Figure 6.
JUSTIFICATION	The activity diagrams of Figure 6 are the same as in the FW Profile definition of [1].
IMPLEMENTATION	The Start operation is implemented by function `FwRtStart`. The Stop operation is implemented by function `FwRtStop`.
VERIFICATION	The verification of this requirement is done in section C.3.

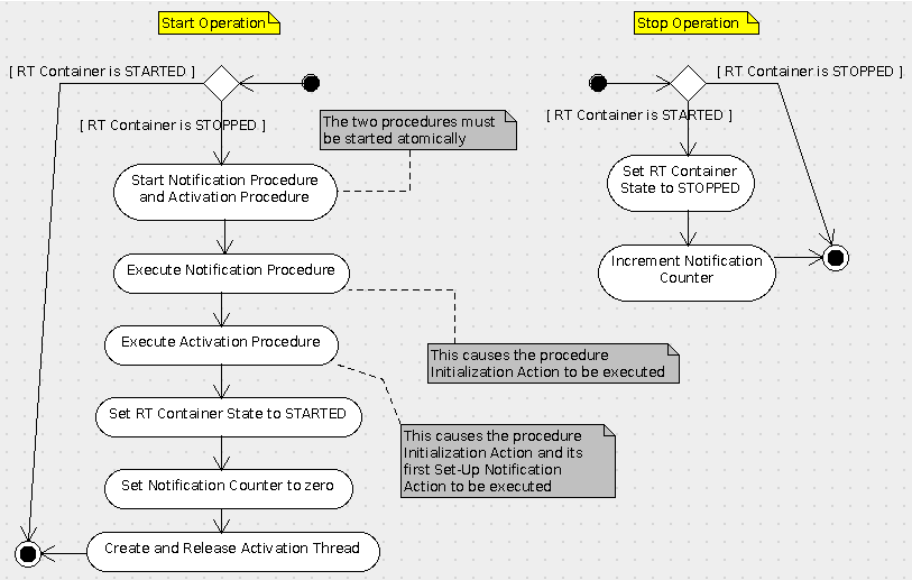

Fig. 6: RT Container Start/Stop Behaviour

5.5 Notification Requirements

FW-5.5.1/T	**Notification Interface**
REQUIREMENT	The C1 Implementation shall provide an interface through which a RT Container can be notified.
JUSTIFICATION	The "Notify" operation is defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the RT container concept of the FW Profile.
IMPLEMENTATION	The Notification interface is implemented in `FwRtCore.h` by function `FwRtNotify`.
VERIFICATION	The Test Cases in the Test Suite send notification commands to RT Containers through the functions of the `FwRtCore.h` module and this module has 100% coverage (except for branches entered in case of POSIX system call errors).

FW-5.5.2/T	Notification Behaviour
Requirement	The C1 Implementation shall implement the "notify" operation to execute the Notification Procedure (see next requirement).
Justification	The "notify" operation is one of the three operations defined by the FW Profile. The intended use of the C1 Implementation is to support the implementation of the RT container concept of the FW Profile.
Implementation	The notification behaviour is implemented in function `FwRtNotify`.
Verification	Test Case `FwRtTestCaseRunDefault1` in the Test Suite verifies that execution of function `FwRtNotify` results in the execution of the Notification Procedure.

FW-5.5.3/T	Notification Procedure
Requirement	The Notification Procedure of a RT Container shall implement the behaviour shown in the activity diagram in the left-hand side of figure 7.
Justification	The activity diagram of figure 7 is the same as in the FW Profile definition.
Implementation	The implementation of the notification procedure is split into two locations: (a) the initialization part (up to the first decision node) is implemented in function `FwRtStart` where the procedure is started and then executed for the first time; (b) the loop is implemented in function `ExecNotifProcedure` in module `FwRtCore.c`.
Verification	Verification of this requirement is done in appendix E where it is shown that every branch of the activity diagram of figure 7 is covered by at least one Test Case in the Test Suite.

FW-5.5.4/T	Activation Procedure
Requirement	A RT Container shall implement an Activation Procedure with the behaviour shown in the activity diagram in the right-hand side of figure 7.
Justification	The activity diagram of figure 7 is the same as in the FW Profile definition.

IMPLEMENTATION	The implementation of the activation procedure is split into two locations: (a) the initialization part (up to the first decision node) is implemented in function `FwRtStart` where the procedure is started and then executed for the first time; (b) the loop is implemented in function `ExecActivProcedure` in module `FwRtCore.c`.
VERIFICATION	Verification of this requirement is done in appendix E where it is shown that every branch of the activity diagram of figure 7 is covered by at least one Test Case in the Test Suite.

FW-5.5.5/T	**Activation Thread**
REQUIREMENT	A RT Container shall encapsulate a thread (the "Activation Thread") to execute the behaviour shown in listing 1.
JUSTIFICATION	The behaviour shown in listing 1 is the same as the behaviour defined in reference [1] for the Activation Thread.
IMPLEMENTATION	The Activation Thread behaviour is implemented in function `ExecActivThread` in module `FwRtCore.c`.
VERIFICATION	Verification of this requirement is done in appendix E where it is shown that every branch of the pseudo-code in listing 1 is covered by at least one Test Case in the Test Suite.

```
1  while true do {
2    wait until Notification Counter is greater than 0;
3    decrement Notification Counter;
4    execute Activation Procedure;
5
6    if (Activation Procedure has terminated) then {
7      put RT Container in STOPPED state;
8      execute Notification Procedure;
9      break;
10   }
11
12   if (RT Container is in state STOPPED) then {
13     execute Activation Procedure;
14     execute Notification Procedure;
15     break;
16   }
17 }
```

Listing 1: Pseudo-code of Activation Thread

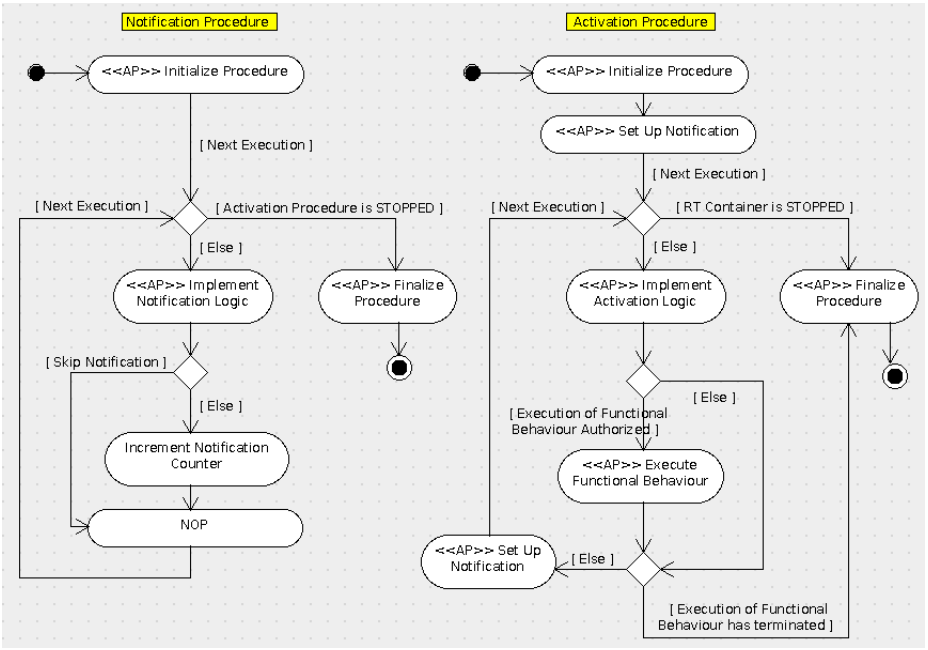

Fig. 7: Notification and Activation Procedures

FW-5.5.6/T	**Wait for Activation Thread Termination**
REQUIREMENT	The C1 Implementation shall offer a function to let a user wait for the termination of a container's Activation Thread.
JUSTIFICATION	One of the usage constraints of a RT Container (see list in table of section 5.4 of reference [1]) requires that a container only be re-started after its Activation Thread has terminated. It is therefore convenient to have a function which waits until the Activation Thread has terminated execution.
IMPLEMENTATION	Function `FwRtWaitForTermination` in module `FwRtCore.h` implements a blocking wait for the termination of the Activation Thread of a RT container.
VERIFICATION	Function `FwRtWaitForTermination` is verified in, among others, the following Test Cases of the Test Suite: `FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, and `FwRtTestCaseStressRun1`.

5.6 Access Requirements

FW-5.6.1/R	Access to Activation Procedure
Requirement	A RT Container shall not allow its users to perform start/stop/execute operations on its Activation Procedure.
Justification	The properties guaranteed by a RT Container (see list in table of section 5.4 of reference [1]) are conditional upon the container's user not having access to the Activation Procedure.
Implementation	The `FwRtCore.h` module does not offer any wrapper functions for starting, stopping, or executing the Activation Procedure (but note that the user has access to the procedure descriptor through the RTD).
Verification	See requirement implementation.

FW-5.6.2/R	Access to Notification Procedure Start/Stop
Requirement	A RT Container shall not allow its users to perform start/stop operations on its Notification Procedure.
Justification	The properties guaranteed by a RT Container (see list in table of section 5.4 of [1]) are conditional upon the container's user not having access to the start/stop interface of the Notification Procedure.
Implementation	The `FwRtCore.h` module does not offer any wrapper functions for starting or stopping the Notification Procedure (but note that the user has access to the procedure descriptor through the RTD).
Verification	See requirement implementation.

FW-5.6.3/R	Access to Notification Procedure Execution
Requirement	A RT Container shall not allow its users to execute its Notification Procedure other than through the Notify operation.
Justification	The properties guaranteed by a RT Container (see list in table of section 5.4 of [1]) are conditional upon the container's user not having access to the execute interface of the Notification Procedure other than through the Notify operation.
Implementation	The `FwRtCore.h` module does not offer any wrapper function for executing the Notification Procedure other than `FwRtNotify` (but note that the user has access to the procedure descriptor through the RTD).

VERIFICATION	See requirement implementation.

5.7 Error Handling Requirements

FW-5.7.1/T	**Error Code**
REQUIREMENT	The RTD shall store the code of the last error encountered during the RTD configuration process or during the processing of notification requests.
JUSTIFICATION	The C1 Implementation targets embedded mission-critical applications where there normally is a need to periodically monitor the integrity of an application. The embedded character of the application, however, also means that memory resources are often limited and it may consequently not be possible to maintain a log of all errors. This requirement represents a compromise between these two needs in the sense that it allows an application to check whether an error has occured with only minimal storage requirements.
IMPLEMENTATION	The C1 Implementation defines both an error code (in field `errCode` of the RTD) and a set of error states (see type `FwRtState_t`) which, taken together, identify the last error condition encountered by a RT Container.
VERIFICATION	The Test Cases `FwPrTestCaseSetAttr1` and `FwPrTestCaseSetAction1` verify the reporting of error conditions arising during the configuration process of a RT container. The only other error conditions reported by a RT container are those arising when a POSIX system call is executed. These error conditions are not verified.

FW-5.7.2/T	**Access to Error Code**
REQUIREMENT	The RTD shall provide read-only access to the error code.
JUSTIFICATION	See the justification of the previous requirement.
IMPLEMENTATION	The value of the error code can be read with function `FwRtGetErrCode`. The value of the error state can be read with function `FwRtGetContState`.
VERIFICATION	See verification of the previous requirement.

6 Non-Functional Requirements

This section defines the non-functional requirements of the C1 Implementation. Non-functional requirements impose overall constraints on the use, design, or implementation of the C1 Implementation.

6.1 Coding Requirements

FW-6.1.1/R	**Implementation Language**
Requirement	The C1 Implementation shall be implemented in the C language.
Justification	The C Language is the standard language for embedded applications.
Implementation	All the modules offered by the C1 Implementation are implemented in C.
Verification	See implementation.

FW-6.1.2/T	**Compiler Warning**
Requirement	The C1 Implementation shall not generate any warnings when compiled with the GCC compiler with all warnings enabled.
Justification	Warning may indicate weaknesses in the code or potential error.
Implementation	See verification.
Verification	The C1 Implementation Acceptance Test Procedure (see reference [2]) compiles all source files of the implementation using `gcc` with the option `-Wall`.

6.2 Use Requirements

FW-6.2.1/R	**SMD Internal Structure**
REQUIREMENT	It shall be possible to create, configure and command a state machine without reference to or knowledge of the internal structure of its SMD.
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The hiding of the information related to a state machine simplifies the interface between the modules of the C1 Implementation and the user application. The target applications of the C1 Implementation are embedded applications. Embedded applications often have special requirements which may require changes to the internal structure of the SMD (perhaps to include more information about a state machine). This requirement would allow this to be done without affecting the interface through which the state machine is manipulated.
IMPLEMENTATION	See requirement verification.
VERIFICATION	The functions and macros to create a state machine (in modules `FwSmDCreate.h` and `FwSmSCreate.h`), the functions to configure a state machine (in module `FwSmConfig.h`), and the functions to send a transition command to a state machine (in module `FwSmCore.h`) take as their argument a pointer to the SMD (i.e. an instance of type `FwSmDesc_t`). There is therefore no need for the user to know the internal structure of the SMD to use these functions.

FW-6.2.2/R	**PRD Internal Structure**
REQUIREMENT	It shall be possible to create, configure and command a procedure without reference to or knowledge of the internal structure of its PRD.

JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The hiding of the information related to a procedure simplifies the interface between the modules of the C1 Implementation and the user application. The target applications of the C1 Implementation are embedded applications. Embedded applications often have special requirements which may require changes to the internal structure of the PRD (perhaps to include more information about a procedure). This requirement would allow this to be done without affecting the interface through which the procedure is manipulated.
IMPLEMENTATION	See requirement verification.
VERIFICATION	The functions and macros to create a procedure (in modules `FwPrDCreate.h` and `FwPrSCreate.h`), the functions to configure a procedure (in module `FwPrConfig.h`), and the functions to execute a procedure (in module `FwPrCore.h`) take as their argument a pointer to the PRD (i.e. an instance of type `FwPrDesc_t`). There is therefore no need for the user to know the internal structure of the PRD to use these functions.

FW-6.2.3/R	**RTD Internal Structure**
REQUIREMENT	It shall be possible to create, configure and command a RT container without reference to or knowledge of the internal structure of its RTD.
JUSTIFICATION	The intended use of the C1 Implementation is to provide modules which can be deployed within another application. The hiding of the information related to a RT container simplifies the interface between the modules of the C1 Implementation and the user application. The target applications of the C1 Implementation are embedded applications. Embedded applications often have special requirements which may require changes to the internal structure of the RTD (perhaps to include more information about a RT container). This requirement would allow this to be done without affecting the interface through which the RT container is manipulated.
IMPLEMENTATION	See requirement verification.
VERIFICATION	The functions which manipulate a RT container take as their argument a pointer to the RTD (i.e. an instance of type `FwPrDesc_t`). There is therefore no need for the user to know the internal structure of the RTD to use these functions.

6.3 Resource Requirements

FW-6.3.1/T	**Code Memory Footprint**
REQUIREMENT	The code memory footprint of the C1 Implementation shall be independent of the size and number of state machines, procedures, and RT containers deployed by an application.
NOTE	Ideally, it would be desirable to impose a requirement on the memory occupation of the C1 Implementation. This is not possible because memory occupation depends on the tool chain used to compile an application and on the target processor. This and the next requirement aim to restrict memory occupation in a manner which is independent of the compilation tool chain and of the execution hardware.
JUSTIFICATION	Embedded applications are often memory-constrained.
IMPLEMENTATION	See requirement verification.
VERIFICATION	The C1 Implementation provides a set of functions which create, configure and manipulate a generic state machine or procedure. RT containers are created by direct instantiation of a type defined by the C1 Implementation. There is no code generation facility (neither explicit, nor implicit through the use of macros) which generates *ad hoc* code for each state machine, procedure, or RT container instance or for categories of state machines, procedures, or RT containers. Thus, the code base of the C1 Implementation is fixed and independent of the number and type of state machines, procedures and RT containers instantiated by an application.

FW-6.3.2/T	**SMD Footprint**
REQUIREMENT	The theoretical memory requirement for an SMD instance shall be a linear function of the values of the following attributes of the SMD: number of states, number of choice pseudo-states, number of transitions, number of actions, number of guards.
NOTE	The expression "theoretical memory requirement" refers to the memory requirement of an ideal compiler which packs all the items in the SMD so as to minimize its memory occupation (in reality, some compilers have alignment constraints which increase memory occupation).

JUSTIFICATION	Embedded applications are often memory-constrained. A linear dependency of the data memory requirements on the size of a state machine minimizes the memory requirements (it is not possible to have a less-than-linear dependency because different state machine instances are independent of each other).
IMPLEMENTATION	The memory occupation of an SMD instance is determined by the internal structure of an SMD which is defined by type `struct FwSmDesc` in `FwSmPrivate.h`.
VERIFICATION	The User Manual (see reference [2]) provides a formula to compute the theoretical memory footprint of a single SMD instance (see section "Memory Footprint"). This formula is linear in the number of states, number of choice pseudo-states, number of transitions, number of actions, number of guards. Note that the requirement asks for the memory footprint of an SMD to be proportional to the number of actions and guards in a state machine. This implies that, in a state machine where the same action is used at different points of the state machine (e.g. the same action acts as entry action of several states), one single function is used to implement all instances of the action.

FW-6.3.3/T	**PRD Footprint**
REQUIREMENT	The theoretical memory requirement for a PRD instance shall be a linear function of the values of the following attributes of the PRD: number of action nodes, number of decision nodes, number of control flows, number of actions, number of guards.
NOTE	The expression "theoretical memory requirement" refers to the memory requirement of an ideal compiler which packs all the items in the PRD so as to minimize its memory occupation (in reality, some compilers have alignment constraints which increase memory occupation).
JUSTIFICATION	Embedded applications are often memory-constrained. A linear dependency of the data memory requirements on the size of a procedure minimizes the memory requirements (it is not possible to have a less-than-linear dependency because different procedure instances are independent of each other).
IMPLEMENTATION	The memory occupation of a PRD instance is determined by the internal structure of a PRD which is defined by type `struct FwPrDesc` in `FwPrPrivate.h`.

VERIFICATION	The User Manual (see reference [2]) provides a formula to compute the theoretical memory footprint of a single PRD instance (see section "Memory Footprint"). This formula is linear in the number of action nodes, number of decision nodes, number of control flows, number of actions, number of guards. Note that the requirement asks for the memory footprint of a PRD to be proportional to the number of actions and guards in a procedure. This implies that, in a procedure where the same action is used at different points of the procedure (e.g. the same action is used in several action nodes), one single function is used to implement all instances of the action.

FW-6.3.4/T	**RTD Footprint**
REQUIREMENT	The memory requirement for the part of an RTD instance directly defined by the C1 Implementation shall be independent of the characteristics of the container represented by the RTD.
NOTE	The restriction to the "part of an RTD instance directly defined by the C1 Implementation" is necessary because an RTD also includes POSIX objects (see requirement FW-6.6.2) whose memory requirements are outside the control of the C1 Implementation and cannot therefore be covered by this requirement.
JUSTIFICATION	Embedded applications are often memory-constrained. This requirement helps keep a boundary on memory requirements for RT containers (it ensures that a "large" container only uses as much memory as a "small" one).
IMPLEMENTATION	The memory occupation of an RTD instance is determined by the internal structure of an RTD which is defined by type `struct FwRtDesc` in `FwRtConstants.h`.
VERIFICATION	The User Manual (see reference [2]) provides a formula to compute the theoretical memory footprint of a single RTD instance (see section "Memory Footprint"). This formula is independent of the characterstics of the RT container.

FW-6.3.5/T	**State Machine Execution Time**
REQUIREMENT	The worst-case execution time of a transition command for a state machine shall be independent of the size of the state machine.

NOTE	Ideally, it would be desirable to impose a requirement on the maximum execution time of a state machine. This is not possible because execution time depends on too many exogenous factors. This requirement aims to restrict execution time in a manner which is independent of the compilation/linking tool chain and execution environment.
JUSTIFICATION	Mission critical applications often need to determine statically the worst-case execution time of an application. Independence of the execution time from the size of a state machine facilitates this task.
IMPLEMENTATION	Transitions in state machines are processed by function `FwSmMakeTrans` in module `FwSmCore.h`.
VERIFICATION	Transitions for state machine are processed by function `FwSmMakeTrans` in module `FwSmCore.h`. This function executes some sequential code and then loops to evaluate the triggers and guards associated to all out-going transitions from the current state. The SMD is internally organized in such a way that it is possible to directly identify all out-going transitions from a given state without searching the entire set of transitions in the state machine. Thus, the maximum size of the loop in `FwSmMakeTrans` is equal to the number of out-going transitions from the current state. Hence, the worst-case execution time occurs when processing a transition from the state in a state machine which has the largest number of out-going transitions. The requirement is therefore satisfied if one assumes that the maximum number of out-going transitions from a state is bounded and independent of the size of a state machine.

FW-6.3.6/T	**Procedure Execution Time**
REQUIREMENT	The worst-case time required by an execution request to traverse a single node in a procedure shall be independent of the size of the procedure.

NOTE	When a procedure is executed, zero or more nodes in the procedure are traversed in sequence. The number of nodes thus traversed depends on the values of the procedure guards at the time the procedure is executed. In the absence of information about the value of the procedure guards, it is therefore not possible to bound the total execution time for a procedure. Ideally, it would be desirable to impose a requirement on the maximum execution time of a procedure. This is not possible because execution time depends on too many exogenous factors. This requirement aims to restrict execution time in a manner which is independent of the compilation/linking tool chain and execution environment.
JUSTIFICATION	Mission critical applications often need to determine statically the worst-case execution time of an application. Independence of the execution time from the size of a procedure facilitates this task.
IMPLEMENTATION	Execution requests in a procedure are processed by function `FwPrExecute` in module `FwPrCore.h`.
VERIFICATION	Execution requests for procedures are processed by function `FwPrExecute` in module `FwPrCore.h`. When an action node is traversed, only sequential code is executed and hence the worst-case execution time is bounded and independent of the procedure size. When a decision node is traversed, some sequential code is executed first and then a loop is executed to evaluate the guards associated to all out-going control flows from the current node. The PRD is internally organized in such a way that it is possible to directly identify all out-going control flows from a given node without searching the entire set of control flows in the procedure. Thus, the maximum size of the loop in `FwPrExecute` is equal to the number of out-going control flows from the current node. Hence, the worst-case execution time occurs when processing a transition from the decision node in a procedure which has the largest number of out-going control flows. The requirement is therefore satisfied if one assumes that the maximum number of out-going control flows from a node is bounded and independent of the size of a procedure.

6.4 Concurrency Requirements

FW-6.4.1/R	Use in Concurrent Environment
REQUIREMENT	It shall be possible for several threads to use the state machine and procedure modules of the C1 Implementation to manipulate multiple SMDs or PRDs without risks for their integrity.
NOTE	The situation covered by this requirement is one where several threads are calling the state machine or procedure functions defined by the C1 Implementation to configure or use different SMDs/PRDs (each thread is manipulating a different SMD or PRD). Clearly, if several threads tried to manipulate the same SMD or PRD, conflicts might arise. These conflicts can only be resolved by the user of the C1 Implementation by building protections into his code.
JUSTIFICATION	Embedded applications are often multi-threaded.
IMPLEMENTATION	See verification.
VERIFICATION	The functions which create, configure and manipulate state machines and procedures do not use any global data structure (they operate on the SMD and PRD instances which are passed to them as an argument). There is therefore no danger for the integrity of an SMD or PRD from multi-threaded access to the C1 Implementation functions. Note that when a state machine or a procedure are extended, their base descriptor is shared between the base state machine or procedure and all its children. The base descriptor however is only accessed in read-mode and hence concurrency poses no danger to its integrity.

FW-6.4.2/T	Concurrent Use of RT Containers
REQUIREMENT	The operations to start, stop and notify a RT Container shall be implemented to be thread-safe.
JUSTIFICATION	RT Container encapsulate an internal thread (the Activation Thread). The user operations to start, stop, and notify a container are in potential conflicts with this internal thread.
IMPLEMENTATION	The start, stop and notify operations are implemented in functions `FwRtStart`, `FwRtStop` and `FwRtNotify` in the `FwRtCore.h` module. These operation use the mutex associated to each container to ensure access in mutual exclusion.

VERIFICATION	The test cases `FwRtTestCaseStressRun1` to `FwRtTestCaseStressRun6` demonstrate operation of a RT container in a multi-threaded environment.

6.5 Verification Requirements

FW-6.5.1/T	**Test Coverage**
REQUIREMENT	The C1 Implementation shall be provided with a Test Suite offering 100% statement, branch and condition coverage (with the exception of code covering the failure of system calls).
NOTE	The term "system calls" covers calls to the `malloc` function in the state machine and procedure modules and to POSIX functions in the RT container module.
JUSTIFICATION	The level of coverage provided by the requirement is that typically used in mission-critical applications. The exclusion of the error branches entered when a system call fails is justified by the difficutly of simulating a system call failure without instrumenting the code.
IMPLEMENTATION	The Test Suite is implemented in a set of Test Cases defined in `FwSmTestCases.h` (for the state machine module), in `FwPrTestCases.h` (for the procedure module), and in `FwRtTestCases.h` (for the RT container module). The `main` program for the Test Suite is in `FwTestSuite.h`.
VERIFICATION	The Acceptance Test Procedure of the C1 Implementation (see [2]) uses the `gcov` tool to measure the statement and branch coverage of the Test Suite. Note that the C1 Implementation does not use any boolean expressions in the decision points of the code (e.g. in the `if` clauses). Decision are always taken on the basis of the outcome of the evaluation of a single primitive Boolean condition. Hence, branch coverage implies condition coverage.

FW-6.5.2/T	**Stress Testing of RT Containers**
REQUIREMENT	The Test Suite for the C1 Implementation shall include stress tests for the RT Containers.
NOTE	The term "stress test" designates tests where a very large number of tests are performed on a certain entity using sequences of pseudo-random conditions and pseudo-random inputs.
JUSTIFICATION	Operation of a RT Container involves interaction of at least two threads (the Activation Threa and the user thread which sends the notification requests to the container). Stress test help explore all possible interaction conditions for the two threads and increase confidence in their correct implementation.

Implementation	The Test Suite implements six stress test cases in `FwRtTestCases.h`. Each stress test case consists of a loop of 10000 operations on a RT container.
Verification	See implementation.

6.6 Dependency Requirements

FW-6.6.1/R	External Libraries
Requirement	The state machine and procedure part of the C1 Implementation shall not require any external library other than C's `stdlib`.
Justification	Minimization of dependencies on external libraries helps minimize the memory footprint of the application using the C1 Implementation and facilitates its qualification. The `stdlib` is likely to be used in any C application and hence is accepted.
Implementation	See verification.
Verification	Inspection of the C1 Implementation files shows that no other library than `stdlib` is used. The compilation and linking process for the Test Suite shows that no other libraries need be linked.

FW-6.6.2/R	Use of POSIX-Compliant Library
Requirement	The C1 Implementation shall rely on a POSIX-compliant library to implement any real-time services it needs for the implementation of RT Containers.
Justification	RT containers need real-time services to implement and interact with the Activation Thread. The POSIX standard is the most widely used in the C and embedded community.
Implementation	The `FwRtCore.h` and `FwRtConfig.h` modules include the `pthread` library which is a POSIX-compliant implementation of threading facilities.
Verification	See requirement implementation.

A Implementation of FW Profile Concepts

The State Machine, Procedure, and RT Container concepts in the FW Profile are defined in terms of their *elements* and in terms of the *operations* which can be performed upon them. This appendix shows how each element and each operation defined by the FW Profile is mapped to a data structure or a function in the C1 Implementation. The information provided in this appendix therefore demonstrates that the C1 Implementation properly covers the State Machine and Procedure Concepts of the FW Profile.

A.1 State Machine Concept

A FW Profile State Machine is defined in terms of its *elements* (see section 4.2 of [1]) and in terms of its *behaviour* (see section 4.3 of [1]). The state machine behaviour is in turn defined in terms of the three *operations* which can be performed upon a state machine.

The State Machine Descriptor or SMD is the data structure which represents a state machine in the C1 Implementation. It is defined in the `FwSmPrivate.h` header file of the C1 Implementation. Table 1 shows how each state machine element is represented within the SMD.

Table 2 shows how each state machine operation is mapped to a function in the `FwSmCore.h` header file.

Table 1: Mapping of SM Elements to Data Structures in the SMD

Element	Implementation in the SMD
Initial Pseudo-State	The Initial Pseudo-State (IPS) is not directly represented in the C1 Implementation data structures. A single "stopped pseudo-state" is used to represent both the Initial and the Final Pseudo-States in the state machine transitions. The transition out of the IPS is the first transition in the Transition Array of an SMD.
Proper State	Proper states are mapped to variables of type `SmPState_t`. The states of a state machine are held in the State Array in the SMD.
State Transition	State transitions are mapped to variables of type `SmTrans_t`. The transitions in a state machine are held in the Transition Array in the SMD.
Choice Pseudo-State	Choice pseudo-states are mapped to variables of type `SmCState_t`. The choice pseudo-states of a state machine are held in the Choice Pseudo-State Array in the SMD.

Element	Implementation in the SMD
Final Pseudo-State	The Final Pseudo-States (FPS) are not directly represented in the C1 Implementation data structures. A single "stopped pseudo-state" is used to represent both the Initial and the Final Pseudo-States in the state machine transitions.
Execution Counters	The Execution Counters are mapped to fields `smExecCnt` (State Machine Execution Counter) and `stateExecCnt` (State Execution Counter) in the SMD.

Table 2: Mapping of SM Operations to Functions in `FwSmCore.h`

Operation	Implementation in `FwSmCore.h`
Start	Function `FwSmStart`
Stop	Function `FwSmStop`
Transition Command	Function `FwSmMakeTrans` and (for the Execute command only) function `FwSmExecute`

A.2 Procedure Concept

A FW Profile Procedure is defined in [1] in terms of its *elements* and of its *behaviour*. The procedure behaviour is in turn defined in terms of the four operations which can be performed upon a procedure. Table 3 shows how each procedure element is mapped to a data structure in the `FwPrPrivate.h` header file of the C1 Implementation and table 4 shows how each operation is mapped to a function in the `FwPrCore.h` header file.

Table 3: Mapping of Procedure Elements to Data Structures in the PRD

Element	Implementation in the PRD
Initial Node	The Initial Node is not directly represented in the C1 Implementation data structures. A single "stopped node" is used to represent both the initial and final node in the procedure control flows. The control flows out of the initial node is the first control flow in the Control Flow Array of a PRD.
Action Node	Action nodes are mapped to variables of type `PrANode_t`. The action nodes of a procedure are held in an Action Node Array in the PRD.
Control Flow	Control Flows are mapped to variables of type `PrFlow_t`. The control flows in a procedure are held in a Control Flow Array in the PRD.
Decision Node	Decision Nodes are mapped to variables of type `PrDNode_t`. The decision nodes of a procedure are held in a Decision Node Array in the PRD.
Final Node	The Final Nodes are not directly represented in the C1 Implementation data structures. A single "stopped node" is used to represent both the initial and the final node in the procedure control flows.
Execution Counters	The Execution Counters are mapped to fields `prExecCnt` (Procedure Execution Counter) and `nodeExecCnt` (Node Execution Counter) in the PRD.

Table 4: Mapping of Procedure Operations to Functions in `FwPrCore.h`

Operation	Implementation in the Procedure Module
Start	Function `FwPrStart`
Stop	Function `FwPrStop`
Execute	Function `FwPrExecute`
Run	Function `FwPrRun`

A.3 RT Container Concept

A FW Profile RT Container is defined in [1] in terms of its *elements* and of its *behaviour*. The container behaviour is in turn defined in terms of the three operations which can be performed upon a container and of the behaviour of its elements (the Activation Thread, the Activation Procedure and the Notification Procedure). Table 5 shows how each container element is mapped to a function in the `FwRtCore.h` header file of the C1 Implementation and table 6 shows how each operation is mapped to a function in the `FwRtCore.h` header file.

Table 5: Mapping of RT Container Elements to Functions

Element	Implementing Function
Activation Thread	The Activation Thread is implemented by a POSIX thread which is stored in field `pThread` of the RTD. The thread behaviour is implemented in function `ExecActivThread`.
Activation Procedure	The behaviour of the Activation Procedure is implemented partly in function `FwRtStart` (this function implement the initialization action of the procedure) and partly in function `ExecActivProcedure` (this function implements the loop part of the procedure).
Notification Procedure	The behaviour of the Notification Procedure is implemented partly in function `FwRtStart` (this function implement the initialization action of the procedure) and partly in function `ExecNotifProcedure` (this function implements the loop part of the procedure).

Table 6: Mapping of RT Container Operations to Functions in `FwRtCore.h`

Operation	Implementation in RT Container Module
Start	Function `FwRtStart`
Stop	Function `FwRtStop`
Notify	Function `FwPrNotify`

B Error Checks

Table 7 lists the configuration errors which are detected by the configuration functions of the state machine module and, for each error, it identifies the test case where the error situation is simulated and its detection is verified.

Similarly, table 8 lists the configuration errors which are detected by the configuration functions of the procedure module and, for each error, it identifies the test case where the error situation is simulated and its detection is verified.

Table 9 lists the error situations which are detected during processing of transition commands for state machine or execution commands for procedures and, for each error, it identifies the test case where the error situation is simulated and its detection is verified.

Errors are reported by setting the error code field of the SMD or PRD. The first column in the tables lists the error code corresponding to each error situation.

Table 7: Verification of Configuration Errors Detected in `FwSmConfig.h`

Error Code	Description of Error	Test Case
`smNullPState`	There is an undefined state in a state machine	`FwSmTestCaseCheck1`
`smNullCState`	There is an undefined choice pseudo-state in a state machine	`FwSmTestCaseCheck2`
`smNullTrans`	There is an undefined transition in a state machine	`FwSmTestCaseCheck3`, `FwSmTestCaseCheck4`, `FwSmTestCaseCheck5`, `FwSmTestCaseCheck6`, `FwSmTestCaseCheck9`
`smIllStateId`	A state is added to a state machine with an illegal (out-of-range) identifier	`FwSmTestCaseConfigErr1`, `FwSmTestCaseConfigErr2`, `FwSmTestCaseDerEmbed1`
`smIllChoiceId`	A choice pseudo-state is added to a state machine with an illegal (out-of-range) identifier	`FwSmTestCaseConfigErr1`, `FwSmTestCaseConfigErr2`
`smStateIdInUse`	A state is added twice to the same state machine	`FwSmTestCaseConfigErr1`, `FwSmTestCaseConfigErr2`
`smChoiceIdInUse`	A choice pseudo-state is added twice to the same state machine	`FwSmTestCaseConfigErr1`, `FwSmTestCaseConfigErr2`

Error Code	Description of Error	Test Case
smUndefinedTransSrc	A transition is added to a state machine with a source (either a state or a choice pseudo-state) which has not yet been defined	FwSmTestCaseCheck11
smIllegalPDest	A transition is added to a state machine with an illegal (out-of-range) state destination	FwSmTestCaseCheck10
smIllegalCDest	A transition is added to a state machine with an illegal (out-of-range) choice pseudo-state destination	FwSmTestCaseCheck12
smIllNOfOutTrans	A choice pseudo-state is added to a state machine with less than 1 out-going transitions	FwSmTestCaseCheck11
smIllTransSrc	A transition is added to a SM with a source (either a state or a choice pseudo-state) which is either not defined or is a proper state which was defined with zero out-going transitions	FwSmTestCaseCheck7, FwSmTestCaseCheck14
smTooManyTrans	A transition from a certain source (either a state or a choice pseudo-state) is added to a state machine but there isn't space for it in the transition array of the state machine descriptor	FwSmTestCaseConfigErr1, FwSmTestCaseConfigErr2, FwSmTestCaseCheck8
smTooManyOutTrans	A state or choice pseudo-state is added to a state machine which has more out-going transitions than fit into the transition array of the state machine descriptor	FwSmTestCaseCheck16

Error Code	Description of Error	Test Case
smTooManyActions	The number of actions added to the state machine exceeds the number of actions declared when the state machine descriptor was created	FwSmTestCaseCheck17, FwSmTestCaseCheck18
smTooManyGuards	The number of guards added to the state machine exceeds the number of guards declared when the state machine descriptor was created	FwSmTestCaseCheck18
smTooFewActions	The number of actions added to the state machine is smaller than the number of actions declared when the state machine descriptor was created	FwSmTestCaseCheck19
smNegOutTrans	A state is added with a negative number of outgoing transitions	TestCaseConfigErr1
smUndefAction	The overridden action in a derived state machine does not exist	FwSmTestCaseDerConfigErr1
smUndefGuard	The overridden guard in a derived state machine does not exist	FwSmTestCaseDerConfigErr1
smEsmDefined	The state in a derived state machine to which an embedded state machine is added already holds an embedded state machine	FwSmTestCaseDerEmbed1
smNotDerivedSM	The state machine where an action or a guard is overridden or a state machine is embedded is not a derived state machine.	FwSmTestCaseDerConfigErr1, FwSmTestCaseDerEmbed1

Error Code	Description of Error	Test Case
smUnreachablePState	The state machine has an unreachable state (a state which is not the destination of any transition).	FwSmTestCaseCheck21
smUnreachableCState	The state machine has an unreachable choice pseudo-state (a state which is not the destination of any transition).	FwSmTestCaseCheck22

Table 8: Verification of Configuration Errors Detected in FwPrConfig.h

Error Code	Description of Error	Test Case
prWrongNOfActions	The number of actions in the base procedure is not the same as in the derived procedure	FwPrTestCaseDerCheck4
prWrongNOfGuards	The number of guards in the base procedure is not the same as in the derived procedure	FwPrTestCaseDerCheck4
prIllActNodeId	An action node is added to a procedure with an illegal (out-of-range) identifier	FwPrTestCaseCheck3
prActNodeIdInUse	An action node is added twice to the same procedure	FwPrTestCaseCheck3
prIllDecNodeId	A decision node is added to a procedure with an illegal (out-of-range) identifier	FwPrTestCaseCheck3
prDecNodeIdInUse	A decision node is added twice to the same procedure	FwPrTestCaseCheck3
prTooManyActions	The number of actions added to the procedure exceeds the number of actions declared when the procedure descriptor was created	FwPrTestCaseCheck5

Error Code	Description of Error	Test Case
prTooManyGuards	The number of guards added to the procedure exceeds the number of guards declared when the procedure descriptor was created	FwPrTestCaseCheck5
prNullAction	An action node is defined with a null action	FwPrTestCaseCheck1
prTooManyOutFlows	A node is added to a procedure which has more out-going transitions than fit into the control flow array of the procedure descriptor	FwPrTestCaseCheck3
prIllNOfOutFlows	A choice pseudo-state is added to a procedure with less than 2 outgoing control flows	FwPrTestCaseCheck3
prTooManyFlows	A control flow from a certain source is added to a procedure but there isn't space for it in the control flow array of the procedure descriptor	FwPrTestCaseCheck5
prIllFlowSrc	A control flow is added to a SM with a source which has an illegal value	FwPrTestCaseCheck5
prConfigErr	A configuration error has been detected during the procedure configuration process	FwPrTestCaseCheck6, FwPrTestCaseDerCheck3, FwPrTestCaseDerCheck5
prNullActNode	There is an undefined action node in a procedure	FwPrTestCaseCheck1, FwPrTestCaseCheck6
prNullDecNode	There is an undefined decision node in a procedure	FwPrTestCaseCheck7
prNullFlow	There is an undefined control flow in a procedure	TestCaseCheck8

Error Code	Description of Error	Test Case
prUndefinedFlowSrc	A control flow is added to a procedure with a source (either a state or a source choice pseudo-state) which has not yet been defined	FwPrTestCaseCheck5
prIllegalADest	A control flow is added to a procedure with an illegal (out-of-range) action node destination	FwPrTestCaseCheck9
prIllegalDDest	A control flow is added to a procedure with an illegal (out-of-range) decision node destination	FwPrTestCaseCheck10
prTooFewActions	The number of actions added to the procedure is smaller than the number of actions declared when the procedure descriptor was created	FwPrTestCaseCheck11
prTooFewGuards	The number of guards added to the procedure is smaller than the number of guards declared when the procedure descriptor was created	FwPrTestCaseCheck12
prUndefAction	The overridden action in a derived procedure does not exist	FwPrTestCaseDerCheck2
prUndefGuard	The overridden guard in a derived procedure does not exist	FwPrTestCaseDerCheck2
prNotDerivedPr	The procedure where an action or a guard is overridden or a procedure is embedded is not a derived procedure	FwPrTestCaseDerCheck2
smUnreachableANode	The procedure has an unreachable action node (a node which is not the destination of any control flow).	FwPrTestCaseCheck13

Error Code	Description of Error	Test Case
smUnreachableDNode	The procedure has an unreachable decision node (a node which is not the destination of any control flow).	FwPrTestCaseCheck14

Table 9: Verification of Dynamic State Machine and Procedure Errors

Error Code	Description of Error	Test Case
smTransErr	A state machine transition encounters a choice pseudo-state which has no out-going transitions with a true guard; or a transition encounters a transition which has a choice pseudo-state as both source and destination of the same transition	FwSmTestCaseTransErr1, FwSmTestCaseTransErr2
prFlowErr	A procedure encounters a decision node which has no out-going control flows with a true guard	FwPrTestCaseCheck4

C Verification of Start/Stop Behaviour

This section demonstrates the test coverage of the logic of the Start and Stop commands for state machines, procedures, and RT containers.

C.1 State Machines

Figure 1 defines the behaviour of a state machine in response to a Start command or to a Stop command. Table 10 lists the test cases which verify the "Start Command" behaviour. Each row in the table corresponds to a branch in the Start activity diagram. Similarly, table 11 lists the test cases which verify the Stop behaviour. Each row in the table corresponds to a branch in the "Stop Command" activity diagram. Note that, in both cases, the tables are not comprehensive in the sense that they do not list all test cases which verify a certain branch. They list at least one test case so as to demonstrate coverage of the corresponding behaviour of the Start or Stop command. The tables therefore verify requirements FW-3.4.2 and FW-3.4.3.

Table 10: Verification of Start Behaviour for a State Machine

Branch	Test Case
State Machine is in a Defined State	`FwSmTestCaseStart1`
State Machine is in an Undefined State	`FwSmTestCaseStart1`, `FwSmTestCaseStart2`, `FwSmTestCaseStart3`
Transition Target is a State	`FwSmTestCaseStart1`
Transition Target is a Choice Pseudo State	`FwSmTestCaseStart2`
Target of Selected Transition is a State	`FwSmTestCaseStart2`
Target of Selected Transition is a FPS	`FwSmTestCaseStart2`
Target State has an Embedded State Machine	`FwSmTestCaseStart3`
Target State has no Embedded State Machine	`FwSmTestCaseStart1`

Table 11: Verification of Stop Behaviour for a State Machine

Branch	Test Case
State Machine is in an Undefined State	`FwSmTestCaseStop1`
State Machine is in a Defined State	`FwSmTestCaseStop1`, `FwSmTestCaseStop2`, `FwSmTestCaseStop3`
Current State has an Embedded State Machine	`FwSmTestCaseStop2`, `FwSmTestCaseStop3`
Current State has no Embedded State Machine	`FwSmTestCaseStop1`

C.2 Procedures

Figure 4 defines the behaviour of a procedure in response to a Start command or to a Stop command. Table 12 lists the test cases which verify the "Start Command" behaviour and the "Stop Command" behaviour. Each row in the table corresponds to a branch in the Start-Stop activity diagram. Note that the tables are not comprehensive in the sense that they do not list all test cases which verify a certain branch. They list at least one test case so as to demonstrate coverage of the corresponding behaviour of the Start or Stop command.

Table 12: Verification of Start and Stop Behaviour of a Procedure

Branch	Test Case
Procedure is started from the STOPPED state	`FwPrTestCaseStart1`
Procedure is started from the STARTED state	`FwPrTestCaseStart1`
Procedure is stopped from the STOPPED state	`FwPrTestCaseStop1`
Procedure is stopped from the STARTED state	`FwPrTestCaseStop1`

C.3 RT Containers

Figure 6 defines the behaviour of a RT Container in response to a `Start` command or to a `Stop` command. Table 13 lists the test cases which verify the "Start Command" behaviour and the "Stop Command" behaviour. Each row in the table corresponds to a branch in the Start-Stop activity diagram. Note that the tables are not comprehensive in the sense that they do not list all test cases which verify a certain branch. They just list one test case so as to demonstrate coverage of the corresponding behaviour of the `Start` or `Stop` command.

Table 13: Verification of Start and Stop Behaviour of a RT Container

Branch	Test Case
RT Container started from STOPPED state	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRun1`
RT Container started from STARTED state	`FwRtTestCaseRunDefault1`
RT Container stopped from STOPPED state	`FwRtTestCaseRunDefault1`
RT Container stopped from STARTED state	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRun2`, `FwRtTestCaseRun3`

D Verification of Execution Behaviour

This section demonstrates the test coverage of the transition commanding logic of state machines and of the execution logic of procedures.

D.1 State Machine Transition Commanding

Figures 2 and 3 define the behaviour of a state machine in response to a Transition Command. Table 14 lists the test cases which verify the behaviour in Figure 2. Each row in the table corresponds to a branch in the activity diagram of the figure. Similarly, table 15 lists the test cases which verify the behaviour in Figure 3. Each row in the table corresponds to a branch in the activity diagram. The tables are not comprehensive in the sense that they do not list all test cases which verify a certain branch. They list at least one test case so as to demonstrate coverage of the corresponding behaviour of the transition command activity diagram.

Table 14: Verification of Transition Command Behaviour of Figure 2

Branch	Test Case
State Machine is not in a Defined State	`FwSmTestCaseExecute1`
State Machine is in a Defined State	`FwSmTestCaseExecute1`, `FwSmTestCaseExecute2`, `FwSmTestCaseExecute3`, `FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans2`, `FwSmTestCaseTrans3`, `FwSmTestCaseTrans4`
Transition Command is the "Execute" Command	`FwSmTestCaseExecute1`, `FwSmTestCaseExecute2`, `FwSmTestCaseExecute3`
Transition Command is not the "Execute" Command	`FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans2`, `FwSmTestCaseTrans3`
Current State has an Embedded State Machine	`FwSmTestCaseExecute2`, `FwSmTestCaseTrans3`
Current State has no Embedded State Machine	`FwSmTestCaseExecute1`, `FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans2`

Branch	Test Case
At least one outgoing transition from current state has transition command as trigger	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans2`
No outgoing transition from current state has transition command as trigger	`FwSmTestCaseExecute1`, `FwSmTestCaseExecute2`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans3`
Guard evaluates to true	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans2`
No guard evaluates to true	`FwSmTestCaseTrans1`

Table 15: Verification of Transition Command Behaviour of Figure 3

Branch	Test Case
Transition source is a state	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans3`
Transition source is a choice pseudo-state	`FwSmTestCaseTrans2`, `FwSmTestCaseTrans5`
Transition source state has an Embedded State Machine	`FwSmTestCaseTrans3`, `FwSmTestCaseExecute4`
Transition source state has no Embedded State Machine	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans2`
Transition target is a state	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans3`
Transition target is a choice pseudo-state	`FwSmTestCaseTrans2`
Transition target is a final pseudo-state	`FwSmTestCaseTrans5`
Destination state has an Embedded State Machine	`FwSmTestCaseExecute4`, `FwSmTestCaseTrans4`, `FwSmTestCaseTrans6`

Branch	Test Case
Destination state has no Embedded State Machine	`FwSmTestCaseExecute3`, `FwSmTestCaseSelfTrans1`, `FwSmTestCaseTrans1`, `FwSmTestCaseTrans2`

D.2 Procedure Execution

Figure 5 defines the behaviour of a procedure in response to an execution request. Table 16 lists the test cases which verify the behaviour in the figure. Each row in the table corresponds to a branch in the activity diagram of the figure. The table is not comprehensive in the sense that it does not list all test cases which verify a certain branch. It lists at least one test case for each branch so as to demonstrate coverage of the corresponding behaviour of the activity diagram.

Table 16: Verification of Execution Behaviour of Figure 5

Branch	Test Case
Procedure is Stopped	`FwPrTestCaseExecute1`
Procedure is Started	`FwSmTestCaseExecute2`, `FwPrTestCaseExecute5`, `FwPrTestCaseExecute6`, `FwPrTestCaseExecute7`
Guard is False	`FwSmTestCaseExecute2`
Guard is True	`FwSmTestCaseExecute2`, `FwSmTestCaseExecute3`, `FwSmTestCaseExecute4`, `FwPrTestCaseExecute5`, `FwPrTestCaseExecute6`, `FwPrTestCaseExecute7`
Target of Control Flow is a Final Node	`FwPrTestCaseExecute5`, `FwPrTestCaseExecute6`
Target of Control Flow is an Action Node	`FwSmTestCaseExecute2`, `FwSmTestCaseExecute4`, `FwPrTestCaseExecute5`, `FwPrTestCaseExecute6`
Target of Control Flow is a Decision Node	`FwSmTestCaseExecute3`, `FwSmTestCaseExecute4`, `FwPrTestCaseExecute7`
Target of Control Flow which evaluates to True is a Decision Node	`FwSmTestCaseExecute4`
Target of Control Flow which evaluates to True is an Action Node	`FwSmTestCaseExecute3`, `FwPrTestCaseExecute7`
Target of Control Flow which evaluates to True is a Final Node	`FwSmTestCaseExecute4`, `FwPrTestCaseExecute7`

E Verification of Notification Behaviour

This section demonstrates the test coverage of the Activation and Notification Procedures and of the Activation Thread of RT Containers. Figure 7 defines the behaviour of the two procedures and listing 1 defines the behaviour of the Activation Thread. Tables 17 and 18 list the test cases which verify the behaviour of the two procedures. Table 19 lists the test cases which verify the behaviour of the Activation Thread. Each row in the tables corresponds to a branch in the activity diagrams of the figure or in the pseudo-code of the Activation Thread. The table is not comprehensive in the sense that it does not list all test cases which verify a certain branch. It lists at least one test case for each branch so as to demonstrate coverage of the corresponding behaviour of the activity diagrams or of the Activation Thread. Note that neither the Activation nor the Notification Procedure ever receives a Stop command: the procedures always terminate naturally by reaching their final node.

Table 17: Verification of Notification Procedure of Figure 7

Branch	Test Case
Procedure is Started	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`
Skip Notification	`FwRtTestCaseRun1`
Do Not skip Notification	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
Activation Procedure is Stopped	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
Activation Procedure is not Stopped	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`

Table 18: Verification of Activation Procedure of Figure 7

Branch	Test Case
Procedure is Started	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
Skip Functional Behaviour	`FwRtTestCaseRun1`
Do Not skip Functional Behaviour	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
RT Container is Stopped	`FwRtTestCaseRun1`

Branch	Test Case
RT Container is not Stopped	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
Functional Behaviour is Terminated	`FwRtTestCaseRunDefault1`, `FwRtTestCaseRunNonNullAttr1`, `FwRtTestCaseRun1`
Functional Behaviour is not Terminated	`FwRtTestCaseRun1`

Table 19: Verification of Activation Thread of Listing 1

Branch	Test Case
Loop is exited because Activation Thread has terminated	`FwRtTestCaseRunDefault1`
Loop is executed more than once	`FwRtTestCaseRun2`
Loop is exited because RT Container has stopped	`FwRtTestCaseRun3`

References

[1] Alessandro Pasetti, Vaclav Cechticky: *The FW Profile*. PP-DF-COR-0001, Revision 1.3.0, P&P Software GmbH, Switzerland, 2013

[2] Alessandro Pasetti, Vaclav Cechticky: *The Framework Profile - C1 Implementation User Manual*. PP-UM-COR-0001, Revision 1.2.0, P&P Software GmbH, Switzerland, 2013

