Service Availability™ Forum Application Interface Specification

Software Management Framework SAI-AIS-SMF-A.01.02

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Forum Application Interface Specification (the "Package") found at the URL http://www.saforum.org is generally made available by the Service Availability Forum (the "Copyright Holder") for use in developing products that are compatible with the standards provided in the Specification. The terms and conditions which govern the use of the Package are covered by the Artistic License 2.0 of the Perl Foundation, which is reproduced here.

The Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed.

The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution.

If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty:

THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

1 Document Introduction 13

1.1 Document Purpose 13
1.2 Document's Organization 13
1.3 History 13
1.3.1 New Topics 14
1.3.2 Clarifications 14
1.3.3 Deleted Topics 14
1.3.4 Other Changes 15
1.3.5 Superseded and Superseding Functions 16
1.3.6 Changes in Return Values of API and Administrative Functions 16
1.4 References 16
1.5 How to Provide Feedback on the Specification 17
1.6 How to Join the Service Availability™ Forum 17
1.7 Additional Information 17
1.7.1 Member Companies 17
1.7.2 Press Materials 18

2 Overview 19

2.1 Overview to the Software Management Framework 19
2.1.1 Service Availability Considerations 21
2.2 Scope of the Software Management Framework Specification 21
2.2.1 Scope of the Current Release 22

3 System Description and System Model 23

3.1 Software Management in SA Forum Systems 23
3.2 Software Delivery 26
3.2.1 Software Catalog 26
3.2.1.1 Software Entity 27
3.2.1.2 Software Entity Type 27
3.2.1.2.1 Base Entity Type 27
3.2.1.2.2 Versioned Entity Type and Prototype 28
3.2.1.3 Software Bundle 28
3.2.2 Handling of Software Bundles 29
3.2.2.1 Repository Management 29
3.2.3 Software Installation and Uninstallation 30
3.2.3.1 Ordering of the Operations for Upgrade 31
3.2.3.2 Ordering of the Operations for Recovery 32
3.2.4 Software Bundle Object Class 32
3.2.5 Entity Types File 33
3.3 Software Deployment 33
3.3.1 Upgrade Campaign 33
3.3.1.1 Upgrade Campaign Model 37

3.3.1.1.1 Upgrade Campaign Object Class . . . 37
3.3.1.1.2 Upgrade Procedure Object Class . . . 38
3.3.1.1.3 Upgrade Step Object Class . . . 39
3.3.2 Upgrade Step . . . 39
3.3.2.1 Deactivation Unit . . . 40
3.3.2.2 Activation Unit . . . 41
3.3.2.3 Actions of the Upgrade Step . . . 42
3.3.3 Upgrade Procedure . . . 43
3.3.3.1 Upgrade Scope . . . 43
3.3.3.2 Upgrade Method . . . 43
3.3.3.2.1 Rolling Upgrade . . . 44
3.3.3.2.2 Single-Step Upgrade . . . 44
3.3.3.3 Procedure Execution Level . . . 45
3.3.4 Service Outage . . . 45
3.4 Upgrade Periods . . . 46
3.4.1 Upgrade Procedure Period . . . 46
3.4.2 Upgrade Campaign Period . . . 47
3.5 Upgrade-Aware Entities . . . 48
3.6 Typical Software Management Information Flow . . . 48

4 Failure Detection and Failure Handling . . . 51

4.1 Failure Detection . . . 51
4.1.1 Upgrade Prerequisites . . . 52
4.1.2 Upgrade Step Verification . . . 55
4.1.3 Upgrade Procedure Verification . . . 56
4.1.4 Upgrade Campaign Verification . . . 56
4.1.5 Exit Status . . . 57
4.2 Failure Handling . . . 57
4.2.1 Protective Measures . . . 58
4.2.1.1 Backup . . . 58
4.2.1.2 Upgrade History . . . 59
4.2.1.3 Detection of Asynchronous Failures of AMF Entities . . . 60
4.2.1.4 Handling Persistent Changes During Upgrade . . . 61
4.2.1.4.1 Changes Caused by the Upgrade . . . 61
4.2.1.4.2 Changes Caused by Normal Operation . . . 62
4.2.2 Recovery Operations . . . 62
4.2.2.1 Undoing an Upgrade Step . . . 63
4.2.2.2 Retry of an Upgrade Step . . . 65
4.2.2.3 Rollback . . . 65
4.2.2.3.1 Campaign Rollback . . . 66
4.2.2.3.2 Procedure Rollback . . . 66
4.2.2.3.3 Step Rollback . . . 66
4.2.2.3.4 Failure During Rollback . . . 67
4.2.2.4 Fallback . . . 68
4.2.2.4.1 Rollforward . . . 69

5 State Models . . . 71

5.1 Upgrade Step State Model . . . 72
5.1.1 Initial State . . . 72
5.1.2 Executing State . . . 72
5.1.3 Completed State . . . 72
5.1.4 Undoing State . . . 74
5.1.5 Failed State . . . 74
5.1.6 Undone State . . . 74
5.1.7 Rolling Back State . . . 74
5.1.8 Undoing Rollback State . . . 75
5.1.9 Rolled Back, Rollback Undone, and Rollback Failed States . . . 75
5.2 Upgrade Procedure State Model . . . 75
5.2.1 Initial State . . . 75
5.2.2 Executing State . . . 75
5.2.3 Suspended and Step Undone States . . . 76
5.2.4 Completed State . . . 77
5.2.5 Rolling Back State . . . 77
5.2.6 Rollback Suspended State . . . 79
5.2.7 Rolled Back, Failed, and Rollback Failed States . . . 79
5.3 Upgrade Campaign State Model . . . 79
5.3.1 Initial State . . . 79
5.3.2 Executing State . . . 79
5.3.3 Execution Completed State . . . 82
5.3.4 Error Detected State . . . 82
5.3.5 Suspending Execution State . . . 82
5.3.6 Error Detected in Suspending State . . . 83
5.3.7 Suspended by Error Detected State . . . 83
5.3.8 Execution Suspended State . . . 84
5.3.9 Rolling Back State . . . 84
5.3.10 Rollback Completed State . . . 85
5.3.11 Suspending Rollback State . . . 85
5.3.12 Rollback Suspended State . . . 85
5.3.13 Execution Failed and Rollback Failed States . . . 86
5.3.14 System Backup, Restart, and Fallback Operations . . . 86

6 Upgrade Campaign Specification . . . 87

6.1 Common Elements . . . 87
6.1.1 Action Element . . . 87
6.1.1.1 Administrative Operation . . . 88
6.1.1.2 Configuration Change Bundle . . . 88
6.1.1.3 CLI Command . . . 88
6.1.1.4 Customized Callback Action . . . 88
6.1.1.4.1 Timing of Customized Callback Actions . . . 89
6.2 Campaign Information . . . 90
6.2.1 Campaign Period . . . 90
6.2.2 Configuration Base . . . 90

6.3 Campaign Initialization 91
6.3.1 Required Software Bundles 91
6.3.2 New AMF Entity Types 91
6.3.3 Initialization Actions 92
6.3.3.1 Generic Initialization Action 92
6.3.3.2 Predefined Conditions for Customized Callback Actions 92
6.3.3.2.1 Callback at Campaign Initialization 92
6.3.3.2.2 Callback at Campaign Backup Creation 93
6.3.3.2.3 Callback at Campaign Rollback 93
6.4 Campaign Body 93
6.4.1 Outage Information 94
6.4.2 Upgrade Method Specification 95
6.4.2.1 Specification of Rolling Upgrades 95
6.4.2.1.1 Target Node 96
6.4.2.1.2 Activation Unit Template 96
6.4.2.1.3 Target Entities 99
6.4.2.1.4 Update Template 100
6.4.2.1.5 Upgrade Step of a Rolling Upgrade 101
6.4.2.1.6 Timing of Callback Actions Within the Procedure 101
6.4.2.2 Specification of Single-Step Upgrades 102
6.4.2.2.1 Deactivation Unit Specification 102
6.4.2.2.2 Activation Unit Specification 103
6.4.2.2.3 Symmetric Activation Unit Specification 103
6.4.2.2.4 Upgrade Step of a Single-Step Upgrade 104
6.5 Campaign Wrap-Up 104
6.5.1 Completion of the Upgrade Campaign 105
6.5.2 Committing the Upgrade Campaign 106

7 Entity Types File 107

7.1 Software Bundle 107
7.1.1 XML Schema for Software Bundles 107
7.1.1.1 Bundle Identification 107
7.1.1.2 Bundle Handling Operations 108
7.1.1.3 Schema Summary 108
7.2 AMF Entity Types and their Prototypes 109
7.2.1 Naming and Versioning 109
7.2.2 Other Attributes 111
7.2.3 XML Schema for AMF Entity Prototypes 111
7.2.3.1 Component Prototype 112
7.2.3.1.1 Provided CS Prototypes 112
7.2.3.1.2 Component Category 113
7.2.3.1.3 CLC-CLI Commands 114
7.2.3.1.4 Upgrade Awareness 114
7.2.3.1.5 Software Bundle Reference 115
7.2.3.1.6 Schema Summary 115
7.2.3.2 Component Service Prototype 119
7.2.3.2.1 Schema Summary 119

7.2.3.3 Service Unit Prototype 120
7.2.3.3.1 Schema Summary 121
7.2.3.4 Service Group Prototype 122
7.2.3.4.1 Schema Summary 123
7.2.3.5 Service Prototype 124
7.2.3.5.1 Schema Summary 124
7.2.3.6 Application Prototype 125
7.2.3.6.1 Schema Summary 125

8 Software Management Framework API 127

8.1 Include File and Library Name 128
8.2 Type Definitions 128
8.2.1 Handles Used by the Software Management Framework 128
8.2.2 SaSmfPhaseT 129
8.2.3 SaSmfUpgrMethodT 129
8.2.4 SaSmfOfflineCommandScopeT 130
8.2.5 Types for State Management 131
8.2.5.1 SaSmfCmpgStateT 131
8.2.5.2 SaSmfProcStateT 132
8.2.5.3 SaSmfStepStateT 132
8.2.5.4 SaSmfStateT 133
8.2.5.5 SaSmfEntityInfoT 133
8.2.6 SaSmfCallbackScopeIdT 133
8.2.7 SaSmfCallbackLabelT 134
8.2.8 Label Filters 134
8.2.8.1 SaSmfLabelFilterTypeT 134
8.2.8.2 SaSmfLabelFilterT 135
8.2.8.3 SaSmfLabelFilterArrayT 135
8.2.9 SaSmfCallbacksT 136
8.3 Library Life Cycle 137
8.3.1 saSmfInitialize() 137
8.3.2 saSmfSelectionObjectGet() 139
8.3.3 saSmfDispatch() 141
8.3.4 saSmfFinalize() 142
8.4 Registration and Unregistration of the Scope of Interest 143
8.4.1 saSmfCallbackScopeRegister() 143
8.4.2 saSmfCallbackScopeUnregister() 145
8.5 Upgrade Campaign Progress Signaling and Response 146
8.5.1 SaSmfCampaignCallbackT 146
8.5.2 saSmfResponse() 148

9 Administrative API 151

9.1 Include File and Library Name 151
9.2 Type Definitions 151
9.2.1 SaSmfAdminOperationIdT 151

9.3 Software Management Framework Administrative API 151
9.3.1 SA_SMF_ADMIN_EXECUTE 152
9.3.2 SA_SMF_ADMIN_COMMIT 154
9.3.3 SA_SMF_ADMIN_SUSPEND 155
9.3.4 SA_SMF_ADMIN_ROLLBACK 157

10 SMF UML Information Model 159

10.1 Notes on the Conventions Used in UML Diagrams 159
10.2 DN Formats for Software Management Framework UML Classes 159
10.3 Software Catalog Classes 160
10.4 Upgrade Campaign Model Classes 162
10.4.1 Upgrade Campaign Model Overview 162
10.4.2 Upgrade Campaign, Upgrade Procedure, and Upgrade Step Classes 163
10.4.3 SMF Deactivation Unit, Activation Unit, and Image-Nodes Classes 165

11 Alarms and Notifications 167

11.1 Setting Common Attributes 167
11.2 Software Management Framework Alarms 168
11.3 Software Management Framework Notifications 168
11.3.1 Software Management Framework State Change Notifications 169
11.3.1.1 Upgrade Campaign State Change Notify 169
11.3.1.2 Upgrade Procedure State Change Notify 170
11.3.1.3 Upgrade Step State Change Notify 171

Index of Definitions 173

List of Figures

Figure 1: The Software Management Framework in the SA Forum Ecosystem 25
Figure 2: Basic Information Model of the Software Catalog 26
Figure 3: Upgrade Campaign Activity Diagram 35
Figure 4: Typical Software Management Information Flow for an Upgrade 49
Figure 5: Upgrade Step State Model Diagram 73
Figure 6: Upgrade Procedure State Model Diagram 78
Figure 7: Upgrade Campaign State Model Diagram 81
Figure 8: SMF Bundle Class 161
Figure 9: SMF Upgrade Campaign Status View 162
Figure 10: SMF Upgrade Campaign, Upgrade Procedure, and Upgrade Step Classes 164
Figure 11: SMF Deactivation Unit, Activation Unit, and Image-Nodes Classes 165

List of Tables

Table 1: Reversing Actions Depending on the Upgrade Step 64
Table 2: Reversing Action Depending on the Step Rollback Actions 68
Table 3: Valid Symmetric Activation Unit Specifications Using Templates 99
Table 4: XML Schema Elements of the Software Bundle Specification 109
Table 5: AMF Entity Types Specification 109
Table 6: XML Schema Elements for Component Prototypes Specification 116
Table 7: XML Schema Elements for CS Prototypes Specification 119
Table 8: XML Schema Elements for SU Prototypes Specification 121
Table 9: XML Schema Elements for Service Group Prototypes Specification 123
Table 10: XML Schema Elements for Service Prototypes Specification 124
Table 11: XML Schema Elements for Application Prototype Specification 125
Table 12: Matching Algorithm for Each Filter Type 136
Table 13: DN Formats 159
Table 14: Upgrade Campaign State Change Notify 169
Table 15: Upgrade Procedure State Change Notify 170
Table 16: Upgrade Step State Change Notify 171

1 Document Introduction

1.1 Document Purpose

This document defines the Software Management Framework (SMF) of the Application Interface Specification (AIS) of the Service Availability™ Forum (SA Forum) to support software upgrade in a single SA Forum system. SMF is the software entity that orchestrates the migration of a live system from one deployment configuration to another while ensuring service availability by tight collaboration with the Availability Management Framework of the Service Availability™ Forum.

Typically, the Service Availability™ Forum Software Management Framework specification will be used in conjunction with the Service Availability™ Forum Application Interface Specification and the Service Availability™ Forum Hardware Platform Interface Specification (HPI).

1.2 Document's Organization

Chapter 1 contains the introduction to this document. Chapter 2 provides an overview of software management and the scope of this specification. Chapter 3 describes the functionality of the Software Management Framework and introduces its basic information model and phases. Chapter 4 defines the failure detection and failure handling. The state model for the upgrade is defined in Chapter 5. Chapter 7 provides guidance for software vendors for the use of the entity types file, while Chapter 6 clarifies the usage of the upgrade campaign specification schema. Chapter 8 specifies the upgrade API and Chapter 9 contains the administrative API. Chapter 10 presents the SMF UML Information Model.
Chapter 11 defines the alarms and notifications. An index of definitions used in this document is presented at the end of the document.

1.3 History

The first and only previous release of the Software Management Framework specification is:

SAI-AIS-SMF-A.01.01

This section presents the changes of the current release, SAI-AIS-SMF-A.01.02, with respect to the SAI-AIS-SMF-A.01.01 release. Editorial changes that do not change semantics or syntax of the described interfaces are not mentioned.

1.3.1 New Topics

⇒ In version A.01.02, as an optional optimization of a rolling upgrade procedure, multiple upgrade steps can be executed concurrently. This extension is described

- in a new paragraph on page 44 (Section 3.3.3.2.1) and
- in a new paragraph on page 95 (Section 6.4.2), which introduces the `saSmfProcDisableSimultanExec` attribute of the `rollingUpgrade` element.

  The `saSmfProcDisableSimultanExec` attribute is shown in FIGURE 10 on page 164.

⇒ As the distribution of entities on nodes may not be known, the upgrade campaign specification for single step allows for (a) a template definition and (b) an IMM modify operation. The corresponding changes are:

- additions to Section 6.4.2.2 on page 102,
- entire replacement of Section 6.4.2.2.1 on page 102,
- entire replacement of Section 6.4.2.2.2 on page 103, and the introduction of
- a new Section 6.4.2.2.3 on page 103.

1.3.2 Clarifications

⇒ The usage of the upgrade campaign `configurationBase` attribute has been clarified in the description of prerequisite 5. on page 53 (Section 4.1.1) and in a paragraph on page 90 (Section 6.2.2).

⇒ On page 76 (Section 5.2.2), a sentence has been added to clarify that setting the `freeze` flag has no impact after the procedure verification has started.

⇒ The term prototype (defined in Section 3.2.1.2.2) has been introduced where necessary to distinguish partially specified versioned entity types used by software vendors to describe their product from fully specified versioned entity types expected by, for example, the Availability Management Framework to be able to manage their entities. Appropriate changes were made all across the document.

1.3.3 Deleted Topics

Chapter 12 of the Software Management Framework A.01.01, which was only a “placeholder” for a Management Interface to be provided in future, has been removed.

1.3.4 Other Changes

⇒ As a consequence of placing the Software Management Framework objects under the Software Management Framework service application object in IMM, changes were made

- in a paragraph on page 32 (Section 3.2.4) and
- in Table 13 on page 159.

⇒ In the description of the prerequisites for starting an upgrade campaign (Section 4.1.1), Prerequisite 8. and its description on page 54 have been added to specify that the Software Management Framework must be able to obtain administrative ownership for at least the objects manipulated in the campaign.

⇒ To align with the Availability Management Framework, which supports the specification of a healthcheck type for a proxied component type, the entity types file schema has been modified to allow for it. Additionally, this schema has been extended to allow the specification of the CS prototype through which

- a component of a proxy component prototype can proxy a given component of a proxied component prototype, or
- a component of a container component prototype can contain a given component of a contained component prototype.

For the corresponding changes, refer to a paragraph on page 112 (Section 7.2.3.1.1) and to the description of the `saAware` element on page 113 (Section 7.2.3.1.2).

⇒ The scope of impact of offline operations has been extended to take into account the scope of a CLM node ([5]). This alignment implied changes in Section 7.1.1.2 on page 108 and in Section 7.1.1.3 on page 108. The `SaSmfOfflineCommandScopeT` enumeration type, which was missing in version A.01.01, is now defined in Section 8.2.4 on page 130. This enumeration uses the name “PLM” in the name definitions; however, no assumption is made regarding the correct use of these values with respect to the Platform Management Service.

⇒ An inconsistency between the XML schema for SU prototype and the corresponding description with respect to the number of component instances has been corrected in a paragraph on page 120 (Section 7.2.3.3).

⇒ A typo has been corrected in the `SaSmfCmpgStateT` enumeration in Section 8.2.5.1 on page 131: The name `SA_SMF_CMPG_ROLLBACK_COMMITTED` was misspelled in SMF A.01.01 as `SA_SMF_CMPG_ROLLBACKC_COMMITTED`. The header file for SMF A.01.01 on the SA Forum Web site does not show this typo.

⇒ In Section 8.2.9 on page 136, a ";" has been added to the declaration of the only member of the `SaSmfCallbacksT` structure.

⇒ The `SaSmfAdminOperationIdT` enumeration type was defined differently in the `saSmf.h` file and in the document. The definition in the header file was the intended variant, which is aligned with similar definitions of other AIS Services. Therefore, the definition (Section 9.2.1) in the specification document was aligned with the header file and appropriate corrections have been made as necessary.

⇒ In the `SaSmfSwBundle` class, the `saSmfBundleInstallCmdArgs` attribute has been replaced with the `saSmfBundleInstallOnlineCmdArgs` and `saSmfBundleInstallOfflineCmdArgs` attributes. Similarly, in the same class, the `saSmfBundleRemoveCmdArgs` attribute has been replaced with the `saSmfBundleRemoveOnlineCmdArgs` and `saSmfBundleRemoveOfflineCmdArgs` attributes. These modifications are shown in FIGURE 8 on page 161.

1.3.5 Superseded and Superseding Functions

None

1.3.6 Changes in Return Values of API and Administrative Functions

None

1.4 References

The following documents contain information that is relevant to this specification:

[1] Service Availability™ Forum, Service Availability Interface, Overview, SAI-Overview-B.04.03

[2] Service Availability™ Forum, Application Interface Specification, Availability Management Framework, SAI-AIS-AMF-B.03.01

[3] Service Availability™ Forum, Application Interface Specification, Notification Service, SAI-AIS-NTF-A.02.01

[4] Service Availability™ Forum, Application Interface Specification, Information Model Management Service, SAI-AIS-IMM-A.02.01

[5] Service Availability™ Forum, Application Interface Specification, Cluster Membership Service, SAI-AIS-CLM-B.03.01

[6] Service Availability™ Forum, Hardware Platform Interface Specification, SAI-HPI-B.02.01

[7] Service Availability™ Forum, Information Model in XML Metadata Interchange (XMI) v2.1 format, SAI-XMI-A.03.02.xml.zip

[8] Service Availability™ Forum, SMF Entity Types File XML Schema Definition, SAI-AIS-SMF-ETF-A.01.02.xsd [1]

[9] Service Availability™ Forum, SMF Upgrade Campaign Specification XML Schema Definition, SAI-AIS-SMF-UCS-A.01.02.xsd[1]

[10] CCITT Recommendation X.733 | ISO/IEC 10164-4, Alarm Reporting Function

References to these documents are made by placing the number of the document in square brackets.

1.5 How to Provide Feedback on the Specification

If you have a question or comment about this specification, you may submit feedback online by following the links provided for this purpose on the Service Availability™ Forum Web site (http://www.saforum.org).

You can also sign up to receive information updates on the Forum or the Specification.

1.6 How to Join the Service Availability™ Forum

The Promoter Members of the Forum require that all organizations wishing to participate in the Forum complete a membership application. Once completed, a representative of the Service Availability™ Forum will contact you to discuss your membership in the Forum. The Service Availability™ Forum Membership Application can be completed online by following the pertinent links provided on the SA Forum Web site (http://www.saforum.org).

You can also submit information requests online. Information requests are generally responded to within three business days.

1.7 Additional Information

1.7.1 Member Companies

A list of the Service Availability™ Forum member companies can be viewed online by using the links provided on the SA Forum Web site (http://www.saforum.org).

1. Files SAI-AIS-SMF-ETF-A.01.02.xsd and SAI-AIS-SMF-UCS-A.01.02.xsd are packaged together; they are contained in a zip archive named SAI-AIS-SMF-XSD-A.01.02.zip.

1.7.2 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information. Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies by following the pertinent links provided on the SA Forum Web site (http://www.saforum.org).

2 Overview

2.1 Overview to the Software Management Framework

SA Forum systems are required to provide highly available services to their users over a long period of time during which the systems may undergo changes due to growth and evolution, bug fixes, or enhancement of services. These changes may require addition, removal, replacement, or reconfiguration of hardware or software elements. High service availability requires that such changes cause no (or only minimal) loss of service.

The different kinds of software that execute on an SA Forum system can be classified as firmware, system software (including hypervisors, operating systems, and middleware), and application software. Such software is constituted of binary or interpreted code that can be executed on the system along with some provisioning data that is required for the software to execute properly.

Some SA Forum Services such as the Availability Management Framework [2] are responsible for controlling the execution of the software on the system (e.g. application software, in the case of the Availability Management Framework). Each SA Forum Service defines its own set of logical entities that are used to either (1) represent instances of software execution under its control or (2) describe the management policies and relationships among these various execution instances.

For example, in the case of the Availability Managed Framework, instances of software execution are represented as *components* while relationships among various components and recovery policies are described through *services units*, *service groups*, and so on.

In the context of the Software Management Framework, all these various logical entities that are used to represent and control software execution are designated as **software entities** (or simply **entities,** where there is no ambiguity). For details on software entities, see Section 3.2.1.1.

A collection of software for these software entities is delivered to the SA Forum system in the form of a **software bundle** (see Section 3.2.1.3). The Software Management Framework maintains the information about the availability, the contents, and the deployment of different software bundles in the SA Forum system. The contents of a software bundle are described in terms of the types of software entities it delivers.

An SA Forum system can be characterized by the **deployment configuration**, which consists of the software deployed in the system along with all configured software entities.

Each software entity is configured in the system as a set of information model objects that must be instantiated within the Information Model Management Service [4]. SA Forum Services such as the Availability Management Framework are the implementers of these information model objects. Note that some information model objects (such as *checkpoint* or *event channel* objects) are not software entities, as they do not represent software execution.

Throughout the life time of an SA Forum system, system evolution is reflected by changes in the deployment configuration. The Software Management Framework is also the service in the system that controls such an evolution in a live system by orchestrating the migration from one deployment configuration to another. This migration process, often referred to as an upgrade, is realized following an **upgrade campaign** specification, which is provided in the form of an XML file. For details on the upgrade campaign, see Section 3.3.1.

During this migration, the Software Management Framework (a) maintains the campaign state model, (b) takes measures that enable error recovery, (c) monitors for potential error situations caused by the migration, and (d) deploys error recovery procedures as required. To accomplish all these tasks, the Software Management Framework interacts with the Availability Management Framework [2] and with other AIS Services and SA Forum HPI implementation(s) as necessary.

The Software Management Framework also provides an API for client processes to register their interest in receiving callbacks when a relevant upgrade is initiated in the cluster and as the upgrade progresses through significant milestones.

The Software Management Framework itself defines a set of logical entities represented as information model objects in the Information Model Management Service. The key logical entities implemented by the Software Management Framework are:

- the software bundle, which represents a collection of software and
- the upgrade campaign, which allows the control and monitoring of a campaign execution.

2.1.1 Service Availability Considerations

From the SA Forum perspective, the only case when the upgrade campaign needs to be designed with service availability in considerations is when

1. software entities under the control of the Availability Management Framework are upgraded and when
2. an upgrade interferes with the normal operation of any of the AMF entities.

Applications that use AIS (Application Interface Specification, [1]) Services, but are not managed by the Availability Management Framework are not considered for service availability unless they fall into the second category. Software executions that are not related to any SA Forum specification, such as the operating system or proprietary database solutions are also not considered for service availability.

If the system is SA Forum compliant only at the HPI level (Hardware Platform Interface [6]), from the SA Forum perspective, it does not provide service availability; therefore, service availability does not need to be taken into account during upgrades of its entities either. HPI provides the Firmware Upgrade Management Instrument (FUMI) API set to support upgrades at HPI level.

2.2 Scope of the Software Management Framework Specification

The Software Management Framework specification defines:

- the functionality provided by the Software Management Framework,
- the software management basic information model and the related XML schema,
- the basic concepts that describe upgrade campaigns and the related XML schema and IMM object classes,
- the basic concepts of error handling during upgrade campaigns,
- the set of administrative APIs to control the execution of an upgrade campaign and the related state model, and
- the set of APIs that client processes can use to register their interest in upgrade campaigns and in receiving callbacks.

As a result, the Software Management Framework specification enables one

- to describe the content of some software collection to be delivered to the SA Forum system,
- to maintain an inventory of the software available in the SA Forum system,
- to specify an upgrade campaign designed according to some availability criteria that deploys some software, and
- to control the campaign execution.

The level of service availability that can be provided during the execution of an upgrade campaign is inherently built into the upgrade campaign specification, and it must be considered when designing the upgrade campaign specification. An implementation of the Software Management Framework executes an upgrade campaign specification, as specified in the XML file provided by the campaign designer. The Software Management Framework specification does not require from an implementation that it must take any measures to maintain service availability beyond those measures specified in the campaign itself.

2.2.1 Scope of the Current Release

The current release of the Software Management Framework specification is limited in its scope.
Its primary goal is to enable the upgrade of software entities managed by the Availability Management Framework; therefore, it is limited to the Availability Management Framework's scope in terms of the types entities managed during an upgrade campaign and the administrative operations the Software Management Framework applies to them. Based on this scope, this release also intends to set the direction and a common understanding for software management in SA Forum systems.

The specification of image management is out-of-scope for this release; however, the information that is necessary to describe the Availability Management Framework-related contents of software bundles is included. The format of the information to be delivered with software bundles is defined by an XML schema.

With respect to the upgrade campaign specifications, this release focuses on the specification of upgrade campaigns for AMF entities of a single cluster using the (node-based) rolling and the single-step upgrade methods. The XML schema for such campaigns is also included.

3 System Description and System Model

3.1 Software Management in SA Forum Systems

The main focus of software management in this document is software upgrade. There are two distinguishable phases of a software upgrade:

1. **software delivery**: the delivery of a new (version of the) software to the SA Forum system;
2. **software deployment**: the act of migrating software entities of the current deployment configuration to the desired deployment configuration, which may use newly delivered software.

These two phases can be executed separately in time. However, in case a new software is deployed, the deployment phase requires the successful accomplishment of the software delivery phase.

A new version of a software is developed by a software vendor and it is normally supplied as a group of files stored in packages according to some predefined format, for example, as Linux RPM packages. A package may contain other related information such as its dependency on other packages, configuration of the new software, its relative installation location, and so on. In SA Forum terms, a collection of such dependent packages and any associated files is referred to as a **software bundle** (for details, see Section 3.2.1.3).

A software bundle becomes available in the SA Forum system when it has been delivered to the **software repository**, a storage location associated with the system. From the software repository, a software bundle can be installed in one or more locations within the system to enable the execution of the included software. It is not specified how the software is packaged within software bundles and how the software bundles are delivered to the software repository. The related requirements and assumptions are described in Section 3.2 including the information that describes each software bundle. The contents of a software bundle are described in a descriptor file, called **entity types file** (see Section 3.2.5). The exact format of this file is defined in [8] by an XML schema. This file provides the software vendor's input to help system integration and configuration.

The Software Management Framework maintains information about the availability, the contents, and the deployment of different software bundles in the SA Forum system. This information is part of the information model maintained by the Software Management Framework. The information model contains the types of software entities available in the system, their versions, and references to the bundles that deliv-

ered them. The collection of software bundles available in the SA Forum system and the types they delivered is referred to as the **software catalog** (see Section 3.2.1).

During the **deployment phase**, the SA Forum system is migrated from the current deployment configuration to the desired deployment configuration by executing an upgrade campaign. The **upgrade campaign** describes the steps and procedures of the migration. This document defines the XML schema [9] to be used to specify upgrade campaigns. Its usage is explained in Chapter 6.

A software upgrade may impact entities that provide highly available services and are managed by the Availability Management Framework. The Software Management Framework coordinates its actions with the Availability Management Framework to avoid or minimize any service loss during such a migration to the desired deployment configuration. This coordination is based on the upgrade campaign specification.

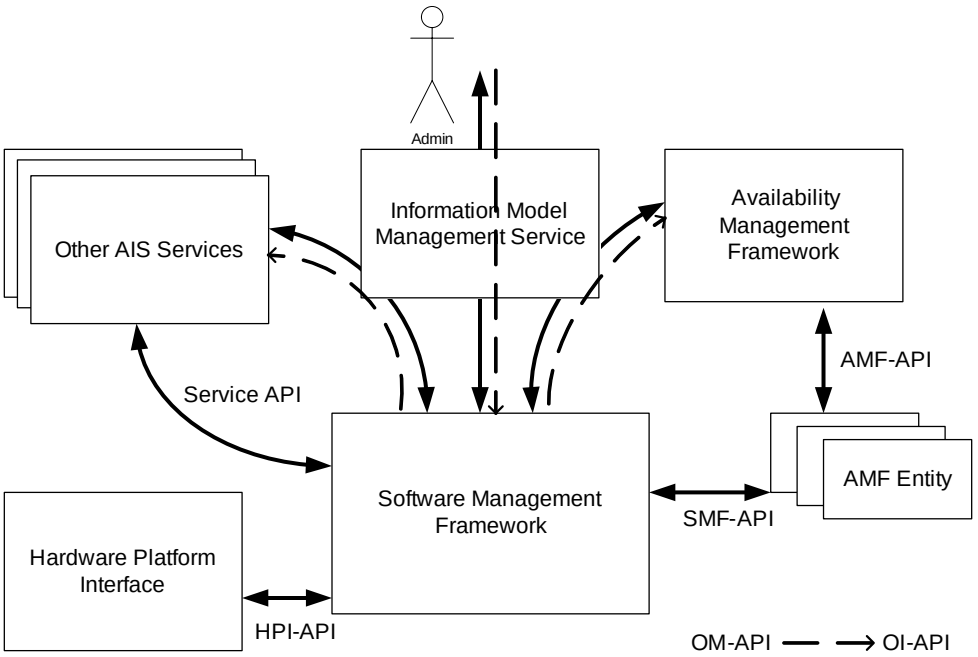
The Software Management Framework takes certain measures to enable error recovery during upgrades, as described in Section 4.2.1. During an upgrade campaign, it continuously monitors for errors according to Section 4.1. It also handles the recovery and repair actions in case of failures during the upgrade campaign, as specified by Section 4.2 and in Chapter 5.

To allow monitoring and the control of an upgrade campaign, the UML model maintained by the Software Management Framework is provided in Section 3.3.1. An upgrade is initiated and controlled by using administrative operations (see Chapter 9) applied to the upgrade campaign object, as defined in Section 3.3 and Chapter 5. The model also reflects the progress of the upgrade campaign according to the state models specified in Chapter 5, facilitating this way the monitoring task.

For applications that need to be aware of an ongoing upgrade campaign within the SA Forum system, the Software Management Framework provides a set of API functions (see Chapter 8) to enable them to coordinate their actions with the progress of the campaign. The Software Management Framework drives these registered applications along the stages defined in the upgrade campaign by invoking the appropriate callback function.

FIGURE 1 summarizes the collaboration of the Software Management Framework with other SA Forum Services within the SA Forum ecosystem.

FIGURE 1 The Software Management Framework in the SA Forum Ecosystem



The Software Management Framework administrative interface is exposed by the Information Model Management Service (IMM, see [4]). Besides, the Software Management Framework also uses IMM to manage the software management information model, which consists of two parts.

- The software catalog, which describes the software available in the system.
- The upgrade campaign model, which allows one to monitor and control the progress of an upgrade campaign within the system.

As a part of the upgrade process, the Software Management Framework uses the administrative API of the Availability Management Framework (AMF, see [2]), which is exposed by the IMM Service to maintain service availability during the upgrade.

The Software Management Framework relies on the SA Forum Notification Service ([3]) to be notified about errors that may be the result of the upgrade process. In particular, the Software Management Framework subscribes to state change notifications issued by the Availability Management Framework to correlate them with the ongoing upgrade process. The Software Management Framework also publishes state change notifications about state changes in the upgrade campaign model.

The Software Management Framework may use other SA Forum Services such as Cluster Membership, Logging, and the Hardware Platform Interface ([6]). It is imple-

mentation-specific when and in what manner these other SA Forum Services are used by the Software Management Framework.

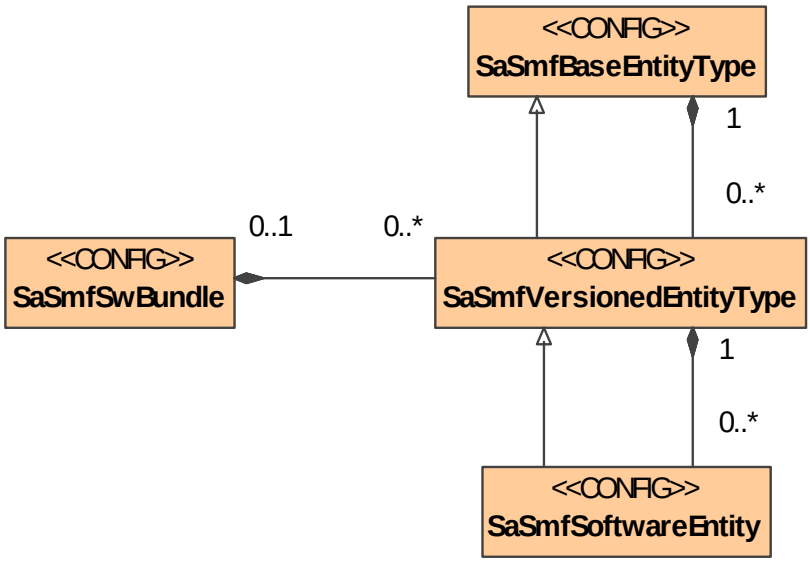
3.2 Software Delivery

This section describes some assumptions that were made with respect to the handling of software packages that compose software bundles, as this is outside the scope of the current document. It also presents the software catalog portion of the information model.

3.2.1 Software Catalog

As shown in FIGURE 2, the **software catalog** portion of the software management information model contains information about the available software entity types in the system, their versions, and references to the software bundles that delivered them and to the entities that deploy them.

FIGURE 2 Basic Information Model of the Software Catalog



This basic information model is further specialized for the different services of the SA Forum system. The current document only presents the model's specialization for the Availability Management Framework entity types (see Section 7.2).

The Software Management Framework uses the Information Model Management Service to store and manage the software catalog built according to specializations of the model.

3.2.1.1 Software Entity

As discussed in Section 2.1, each SA Forum Service defines its own set of logical entities that are used to either

- represent instances of software execution under its control or
- describe the management policies and relationship among these various execution instances.

In the context of the Software Management Framework, all these various logical entities are designated as **software entities**. However, the Software Management Framework does not implement these software entities; instead, it only configures them in terms of adding, removing and modifying them, as specified by an upgrade campaign.

Software entities are configured in the system as information model objects of the Information Model Management Service [4]. SA Forum Services such as the Availability Management Framework are the implementers of these information model objects as appropriate.

3.2.1.2 Software Entity Type

The generalization of similar software entities is called a **software entity type** (or simply a **type**, if the context permits). Any software entity that is present in an SA Forum system and needs to be manipulated by the Software Management Framework must be of certain type.

The software management information model distinguishes two classes of software entity types:

- base entity types and
- versioned entity types.

3.2.1.2.1 Base Entity Type

Base entity types are not directly associated with executable pieces software; instead, they represent a common functionality these pieces of software deliver, each of which is versioned. Thus, the base entity type is a container of versioned entity types that are grouped together by some common functionality.

The base entity type has a name to which all its versions refer.

This document presents the base entity types defined for the SA Forum Availability Management Framework in Section 7.2.

3.2.1.2.2 Versioned Entity Type and Prototype

A base entity type can be implemented by different versions of software. Each such version is represented as a **versioned entity type** (or simply as a **version**, if the context permits).

A versioned entity type expands the base entity type with versioning and version-specific information. It describes the deployment constraints such as the list of compatible and required versions of other versioned entity types. It also defines all the attributes (together with their default values) that are inherited by its instances. Software usually allows for different configurations, which is reflected in allowing vendors only to partially specify their versioned entity types. In other words, they specify versioned entity **prototypes**, from which fully specified versioned entity types are derived for deployment as part of the system or site integration task.

For the Availability Management Framework, this document specifies the versioned entity prototypes in Section 7.2, as they must be used by a software vendor. The Availability Management Framework specification [2], on the other hand, specifies versioned entity types, as they must be used in a running system, which reflects the vendor's information tailored to the specifics of a particular system.

To be deployed, a base entity type must have at least one version delivered to the system. This parent-child relation between a base and versioned types is implicitly defined by their naming. The naming of AMF entity types is specified in Section 7.2.1.

A versioned entity type may be related to an entity type that is not its parent. For example, different versions of a component type provide the same component service type. Such a relationship needs to be specified explicitly as an attribute.

Software bundles deliver versioned entity prototypes.

3.2.1.3 Software Bundle

The **software bundle** of the information model represents a collection of interdependent software packages and associated files that has been delivered to the system's repository. The software bundle is the smallest unit recognized by the Software Management Framework and that is represented in the information model.

A software bundle may deliver different versioned entity (proto)types; however, it will typically not deliver multiple versions of the same base type. The software bundle object is referenced by all the versioned entity types whose software is delivered by the bundle.

3.2.2 Handling of Software Bundles

3.2.2.1 Repository Management

It is assumed that an SA Forum system is associated with a logically single storage that collects all software bundles available in the system. In the software delivery phase, software bundles are made available to the system by copying them to the software repository. The source could be a remote file server, a CD, or some other media. The Software Management Framework specification currently does not specify how software packages and, as a consequence, software bundles are imported to the repository and made available to the system.

When software bundles are delivered to the software repository, they should be verified in an implementation-specific way (for instance, by the use of package handling utilities of an operating system or by other means). This verification should include for each software bundle:

- A check of the integrity and the origin of all software packages that make up the bundle.
- A check that all other packages indicated in the package dependency list are available in the repository or have been delivered at the same time.
- A check that the bundle has been properly added to the repository and can, therefore, be used as an installation source.

When all these operations and checks are successfully completed, the software bundle becomes available for the system and allows one to add an IMM object representing the software bundle to the information model of the software catalog.

If any of these operations fails, the bundle may not be added either to the repository or to the information model.

When the IMM object representing a software bundle is deleted from the information model of the software catalog, the associated bundle becomes unavailable for the SA Forum system and needs to be removed from the software repository. The bundle is removed in an implementation-specific way. This should not happen while the software is still in use within the system.

3.2.3 Software Installation and Uninstallation

Software installation means the creation of an executable form at a location from which the software can be executed on a target node. The source of the installation is a software bundle in the software repository. Saying that the software is installed means that it can be used to instantiate software entities according to their configuration. **Software uninstallation** means the removal of an installed software from such a target location; however, the software bundle remains available in the software repository.

Depending on the nature of the software and the execution environment, the installation and uninstallation may or may not interfere with software entities executing (or which can potentially execute) in the system. Based on this characteristic, the Software Management Framework distinguishes two categories of installation and uninstallation operations: **online** and **offline**.

An **online operation** does not interfere with any software entity in the system, especially not with one that is managed by the Availability Management Framework. An online operation can be executed at any time, as it does not require availability management of entities. The online software installation may be part of the delivery phase. The Software Management Framework may execute such online operations without initiating an upgrade campaign, as a preparation for an upgrade campaign - in case of software installation - or after the completion of the campaign - in case of an uninstallation.

Operations that do not satisfy the above criteria fall into the **offline** category. To prevent any interference, the impacted entity and maybe other entities need to be taken out of service for the time of the operation. Accordingly, an offline operation has a scope of impact that requires availability management. As a result, offline operations may only be executed as part of an upgrade campaign. For proper handling, the minimal scope of impact needs to be indicated by the software vendor.

For each bundle, the following must be specified in form of CLI commands:

- the portions of the installation and uninstallation that can be executed online,
- the portions of the installation and uninstallation that need to be executed offline, together with the minimum scope of impact that must be taken into account at campaign design.

A software bundle is always installed or uninstalled as a whole at a target location. This means that all entity types included in the bundle are installed or uninstalled and become available or unavailable at that target location after the execution of the pair of online and offline CLI commands of the appropriate operation.

3.2.3.1 Ordering of the Operations for Upgrade

The installation and uninstallation operations are executed in the following order when a software bundle is used to upgrade an existing installation at a given target location:

1. online installation of the new software
2. offline uninstallation of the old software
3. offline installation of the new software
4. online uninstallation of the old software

The operations 1. and 4. may be executed outside a campaign, whereas operations 2. and 3. are always executed within the scope of the upgrade campaign.

The successful completion of an operation is a prerequisite for initiating the subsequent operation. Some of these operations may be empty, in which case they are considered to be successful.

The CLI commands used to perform the installation and uninstallation operations must be implemented so that when the same operation is executed on the same target location a second time, the installation status of the software is verified and corrected as necessary.
Example of a second operation on the same target location: the online installation (operation 1.) is executed when the package is delivered. After some time, the relevant upgrade campaign is initiated. During this campaign, the Software Management Framework must be able to re-use operation 1. to verify the already installed software and correct it if necessary.

The operation should succeed in any of the following cases:

- installation exists, and it is correct;
- installation exists, but it was corrupted, and a successful repair was performed;
- installation did not exist; therefore, a new successful installation was performed.

After completion of operation 2., the old software is no longer available for execution at the target location. After completion of operation 4., the old software must be completely removed from the target location.

After completion of operation 3., the new software must be ready for execution at the target location. For example, if an Availability Management Framework component is delivered by this software, the Availability Management Framework should be able to instantiate it using the CLC-CLI command specified for the component.

To speed up any recovery operation, operation 4. is typically not performed before the upgrade campaign has completed.

3.2.3.2 Ordering of the Operations for Recovery

The operations to restore the old software (for example, due to a campaign failure or to other reasons) are performed as follows:

1. online installation of the old software
2. offline uninstallation of the new software
3. offline installation of the old software
4. online uninstallation of the new software

If the old software was not yet removed, operation 1. verifies and corrects the installation of the old software. Operation 4. may or may not be performed until the campaign (in this case, its rollback) completes.

The Software Management Framework mandates that for each bundle each of these operations is specified as a CLI command to be initiated at the target location, that is, in the execution environment of the appropriate cluster node.

3.2.4 Software Bundle Object Class

The software catalog of the information model is populated with objects of the software bundle object class, which is presented in Section 10.3. These objects represent software bundles available in the software repository.

Software bundles must have a unique distinguished name (DN).The relative distinguished name (RDN) has the form of:

```
safSmfBundle=...
```

To minimize name conflicts, the RDN should include a prefix specific to the particular vendor. The stock symbol or the Internet domain name of the company providing the software bundle are examples for such a prefix. This unique DN is used by the Software Management Framework for the software bundle object in the software management information model.

The CLI commands for the bundle handling operations are presented as URIs. In addition, the software bundle object class defines a default timeout that is used to limit the CLI operations.

3.2.5 Entity Types File

The **entity types file** is a target system-agnostic description of the content of a software bundle delivered to an SA Forum system. Each SA Forum-compliant software bundle delivered to the software repository must be associated with an entity types file. One entity types file may refer to multiple software bundles, each of which must be identified according to the XML schema defined in [8] and discussed in Chapter 7. Some of these bundles may already be part of the software repository, and from the identification information, it must be possible to determine their identity. The entity types file also shall specify the CLI commands to be used to install and uninstall each referenced bundle.

The entity types file is the means by which a software vendor shall describe the SA Forum-relevant contents and features of its product and any implementation-specific constraint that needs to be taken into account by a deployer wanting to use the product.

Accordingly, additional information on each software entity prototype needs to be given as necessary to facilitate the following tasks:

- identification of software entities that are potential targets for the upgrade using the software delivered in the bundle;
- verification of the compatibility of the delivered entity prototypes with those currently available in the system and the software entities deploying them;
- proper configuration of those software entities that were selected for upgrade;
- generation of an upgrade campaign specification.

The XML schema for the different entity prototypes that are recognized by the SA Forum Software Management Framework is specified in [8]. This release only considers Availability Management Framework entity types, which are discussed in more details in Section 7.2.

3.3 Software Deployment

3.3.1 Upgrade Campaign

The deployment configuration of an SA Forum system may need to be changed at any time, for example, to tune its performance by creating new instances of an entity type available in the software catalog, by modifying parts of the configuration, or by removing some of the existing instances. In some cases, it is necessary to downgrade entities to an earlier version of their base entity type.

Whenever a new software bundle is delivered to the SA Forum system, it creates the potential for adding new entities of the newly delivered types to the system configuration. If there are already software entities in the system that belong to the earlier version of the base entity type of a version delivered by the bundle, these software entities become upgradable to the new version.

Any of these operations require the migration of the current deployment configuration to a new one, which is performed by the Software Management Framework as an **upgrade campaign** following the instructions of an upgrade campaign specification.

FIGURE 3 presents the major activities comprising an upgrade campaign.

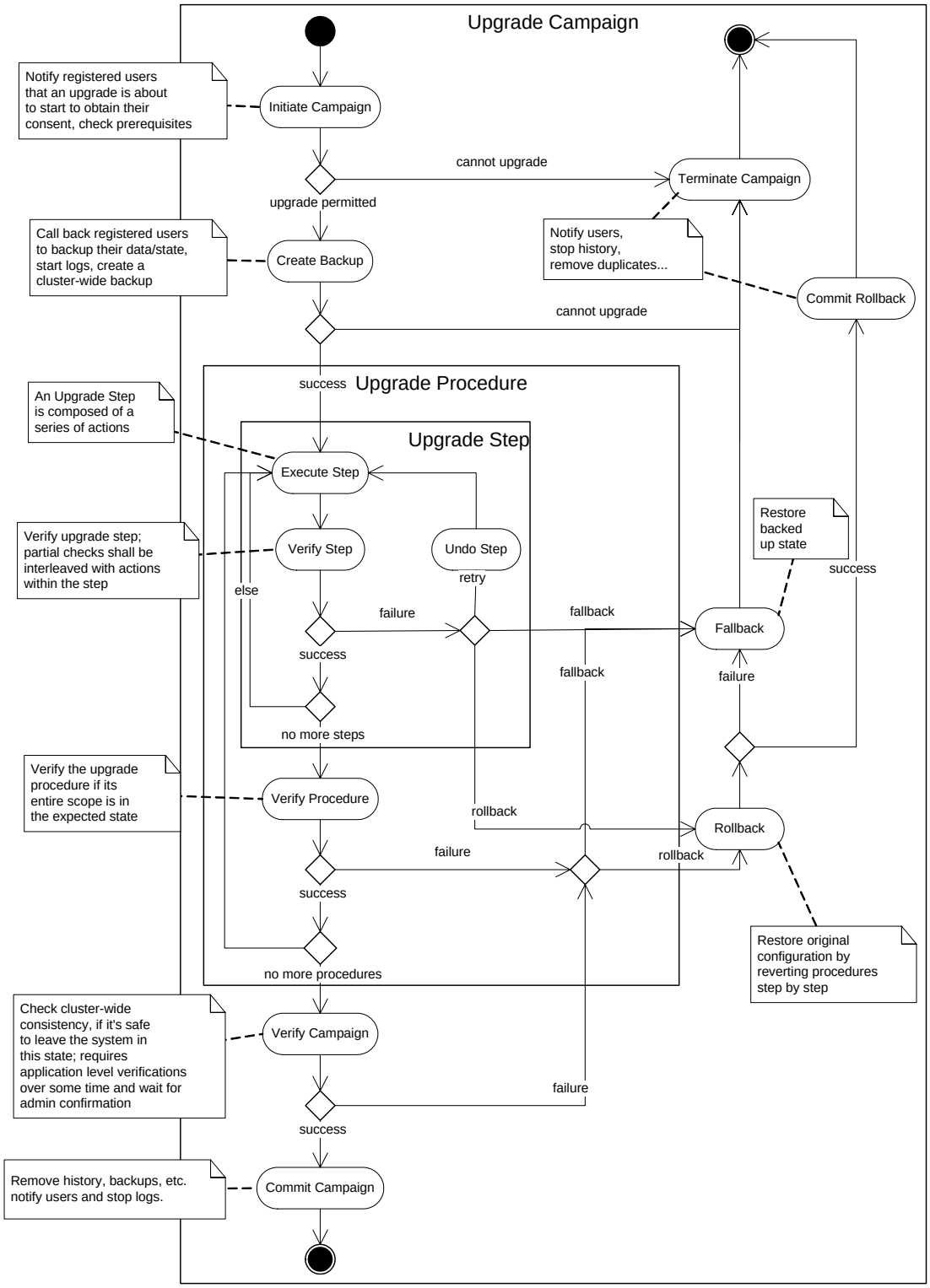
An upgrade campaign is handled in the manner of transactions, and, accordingly, it is initiated and committed or terminated. These operations determine the transaction boundaries.

At the initiation of the upgrade campaign (action `initiate campaign` in FIGURE 3), the Software Management Framework determines whether the upgrade campaign can proceed according to some prerequisites (see details in Section 4.1.1). If it cannot proceed, the upgrade is terminated (`terminate campaign`); otherwise, a cluster-wide backup is created (`create backup`) as described in Section 4.2.1.1.

In the cluster-wide backup, both the system and the applications are requested to save their own information as necessary. With the backup, some logs may also be started to capture permanent changes that occur after the backup (Section 4.2.1.4). These permanent changes will be reapplied to the state restored from the backup in case a fatal failure demands so (Section 4.2.2.4.1). Consequently, if the backup or any of these preparative actions fail for any reason, the upgrade cannot proceed.

If the upgrade proceeds, there are virtually two configurations in effect: the current deployment configuration and a new desired configuration. The Software Management Framework's task is to orchestrate the transition between the two according to the upgrade campaign specification.

FIGURE 3 Upgrade Campaign Activity Diagram



Assume a configured entity of the deployment configuration that needs to be upgraded. This entity must refer to the versioned entity type it instantiates. To upgrade this entity to a new version of its type, the new version must be delivered first. After the upgrade, the configured entity needs to point to this new version, which is the desired configuration. This transition is performed as part of an upgrade campaign when the Software Management Framework has completed all the necessary actions that allow the entity to be instantiated from the new version. These actions are performed in an upgrade step (`execute upgrade step`). An upgrade step may transition a single entity or a set of them (Section 3.3.2) to their new configuration.

In order to maintain availability, an upgrade campaign typically does not migrate all entities of a given type simultaneously; instead, the actions that are necessary to perform the migration are structured and organized into upgrade steps and procedures to distribute them in time in a synchronized way, so that service availability can be maintained by the system. An upgrade procedure iterates the same upgrade step over sets of similar software entities (Section 3.3.3). An upgrade campaign may define one or more upgrade procedures (Section 3.3.1).

The actions are continuously verified (`verify step`). Besides the step level, additional verification may be performed at the procedure and campaign levels (see Section 4.1).

If the verification of an upgrade step fails, it may be retried after undoing the actions of the failed step (Section 4.2.2.2). If retry is not possible, or a procedure verification fails, the campaign should roll back all the way to the start (`complete rollback`), as described in Section 4.2.2.3. In some cases, the rollback is not possible, for instance, when the system is in an unknown faulty state, or when the rollback requires more time than it is available in the current maintenance window. In such cases, a fallback is triggered (Section 4.2.2.4) to restore the state saved during the backup operation. A fallback may also be triggered if a failure occurred during rollback. In most of these cases, the Software Management Framework suspends the campaign when the fault is detected and the administrator has to decide how to proceed.

Once all the upgrade procedures are completed, the entire system is checked for consistency. This check may also involve application level verification. If it is determined that the system is in a consistent operational state, the upgrade campaign is committed and wrapped up (`commit campaign`). These actions include freeing some or all of the resources (for instance, history, logging) that were used during the upgrade campaign.

If the final verification fails, the initial system configuration needs to be restored by rollback, if possible, or by a fallback.

The Software Management Framework does not make the decision about the success of the campaign automatically. The administrator is expected to commit the campaign after checking the information presented by the Software Management Framework, and possibly after performing some manual verifications.

3.3.1.1 Upgrade Campaign Model

An upgrade campaign is specified and provided to the Software Management Framework as an XML file, which is discussed in details in Chapter 6. The schema for the upgrade campaign specification file is contained in [9].

The upgrade campaign specification file is provided by the administrator as an input parameter at the creation of an campaign object in the information model. This campaign object is used as the target object for administrative operations controlling the campaign execution.

From the upgrade campaign specification file, the Software Management Framework completes the upgrade campaign model in the information model, allowing an administrator to also monitor the status of the execution. For the upgrade campaign model, refer to FIGURE 9 in Section 10.4.1.

As discussed in Section 3.3.1, an upgrade campaign is staged in a set of upgrade procedures, each of which is executed as a sequence of upgrade steps.

3.3.1.1.1 Upgrade Campaign Object Class

The upgrade campaign is a configuration object class. An upgrade campaign object must be configured before the campaign can be executed. The upgrade campaign object includes an attribute which specifies the associated upgrade campaign XML file.

The upgrade campaign XML file completely specifies the process of migration of a live system from one deployment configuration to another desired one. The upgrade campaign is composed of an ordered set of upgrade procedures, which are represented as children of the upgrade campaign object.

An upgrade campaign is named (see `upgradeCampaign` element in Chapter 6), and its RDN has the format:

```
safSmfCampaign=<name>
```

Typically, an upgrade campaign can be applied to a system only once successfully. It may also be tied to a given deployment configuration that it should upgrade to the desired one; therefore, it typically cannot be applied if the deployment configuration has changed.

The administrative control of an upgrade is applied to the campaign object, which is a configuration object of the `SaSmfCampaign` class with a single configuration attribute, the location of the upgrade campaign specification file in form of a URI.

The information model object (see FIGURE 10 in Section 10.4.2) reflects the state of the upgrade campaign according to the state model described in Section 5.3 and other significant information about the campaign, such as the time stamp of the IMM content (see Prerequisite 5. on page 53) to which the campaign is relevant, the expected and elapsed execution time (see Section 3.4.2), and any additional information on detected errors.

3.3.1.1.2 Upgrade Procedure Object Class

An upgrade procedure applies the same upgrade step over a set of typically identical groups of entities according to a selected upgrade method. The upgrade method determines the number of upgrade steps, the sequence of their execution, and possibly other constraints that need to be observed during the execution.

Within a campaign, each procedure is named (see `upgradeProcedure` element in Section 6.4) and represented by a procedure runtime object of the class `SaSmfProcedure` in the information model (see FIGURE 10 in Section 10.4.2). The format of the upgrade procedure RDN is:

```
safSmfProcedure=<name>
```

Depending on their execution level, upgrade procedures may be executed simultaneously. Those of different execution levels must be executed sequentially in the order determined by the execution level, starting with the lowest execution level.

An important attribute of the upgrade procedure is the **service outage** (see Section 3.3.4) it is allowed to introduce during execution. This acceptable service outage is expressed as service instances that may be disrupted (become unassigned) for the time of the execution of the upgrade procedure. The service outage is taken

into account when the upgrade campaign is initiated. Once the execution has started, it is not guaranteed that the actual service outage will not exceed this acceptable level.

The information model object also reflects the current state of each procedure in the campaign according to the state model (see Section 5.1.9).

3.3.1.1.3 Upgrade Step Object Class

An upgrade step is a sequence of actions that logically belong together and that are applied to a group of software entities to migrate some or all of them from their current configuration to their intended configuration. An upgrade step may also add new entities to the configuration or remove some entities.

Such a change is typically achieved by deactivating the current set of entities and reactivating the same or a modified set of entities at the end of the upgrade step. Hence, the two sets are referred to as **deactivation** and **activation units** and represent the most important attributes of the upgrade step object.

The Software Management Framework generates an RDN for each upgrade step in the format of

```
safSmfStep=<integer>
```

These objects are runtime objects of the `SaSmfStep` object class (see FIGURE 10 in Section 10.4.2). In addition to the mentioned attributes, they reflect the state according to the state model of Section 5.1.

3.3.2 Upgrade Step

An **upgrade step** is a sequence of actions, each of which is carried out on a set of software entities. These actions logically belong together, as they are necessary to migrate this group of software entities to the desired configuration.

The deactivation and activation units may specify entities that are not targeted directly by the upgrade. This is because some actions of the upgrade step may affect the behavior and availability of entities that are not being upgraded. To manage the availability of these affected entities, they also must be included in the deactivation or activation units as necessary.

3.3.2.1 Deactivation Unit

To perform an upgrade step without service disruption, the affected software entities need to be taken out of service. The collection of these entities is called the upgrade step's **deactivation unit** (see FIGURE 11 in Section 10.4.3).

By definition, the actions of an offline uninstallation and an offline installation (see Section 3.2.3) of a software bundle impact some software entities in the system. The minimum scope of this disturbance is given as an attribute of the software bundle descriptor (see Section 7.1.1.2) for each of the operations. When designing the campaign, these minimum scopes and the target system's characteristics need to be taken into account to determine the set of entities that needs to be deactivated during the upgrade step.

The Software Management Framework deactivates the entities within the deactivation unit by executing the appropriate administrative operations: It first locks each entity listed in the deactivation unit and then terminates them. Therefore, the deactivation unit must list all the entities on which these operations need to be issued. The current document considers only AMF administrative operations: lock, for taking AMF entities out of service and lock-instantiation, for terminating them.

After terminating the entities of the deactivation unit, the offline uninstallation and offline installation operations shall be carried out. Each software bundle that is removed from a set of nodes by the upgrade step is represented by a runtime object of the class `SaSmfImageNodes` (see FIGURE 11 in Section 10.4.3). Software bundles that are installed by the step are presented as objects of the same type but appended to the activation unit information model object.

The campaign designer must consider all actions of the upgrade step when the deactivation unit is specified. In particular, the deactivation unit must be at least the union set of the impacted entities. Some other considerations are also necessary, such as the available set of AMF administrative operations. For instance, a component cannot be deactivated on its own by acting on it directly, but only by deactivating the enclosing service unit. If components of different service units need to be deactivated, all the enclosing service units need to be deactivated; thus, it may happen that in some cases the deactivation of the entire AMF node is more feasible than the deactivation of single service units located on the node.

The deactivation unit may be empty, in which case the upgrade step shall not impact the behavior of any of the entities in the system (at a minimum, no offline package handling operation is required), or the impacted entities do not provide directly or indi-

rectly highly available services, that is, none of the directly or indirectly impacted entities are managed by the Availability Management Framework.

3.3.2.2 Activation Unit

At the end of the upgrade step, some or all of the deactivated entities and the newly added software entities need to be activated or reactivated. The collection of software entities that are put back into service is called the upgrade step's **activation unit** (see FIGURE 11 in Section 10.4.3).

Before this activation or reactivation may happen, all the software installations must be completed for the entities of the activation unit. Each software bundle that is installed on a set of nodes by the upgrade step is represented by a runtime object of the `SaSmfImageNodes` object class (see FIGURE 11 in Section 10.4.3). Also all the information model updates must be applied to reflect the desired configuration for all entities in the activation unit.

For those software entities within the activation unit that have a configuration attribute to indicate the maintenance status of the entity by containing the DN of any ongoing maintenance campaign impacting the entity, and that are being upgraded by the upgrade step, the Software Management Framework must set this attribute to the DN of the upgrade campaign object. If an entity does not have such an attribute, the attribute must be set for the closest encapsulating entity as applicable. For example, the B.03.01 release of the Availability Management Framework specification defines such an attribute only for service units (`saAmfSUMaintenanceCampaign`). This attribute must be set by the Software Management Framework to the upgrade campaign DN for a particular service unit if any of the following apply:

- the service unit itself is being upgraded by the upgrade step;
- any component within the service unit is targeted by the upgrade step; or
- the AMF node hosting the service unit is specified as the target of the upgrade step.

The Software Management Framework activates the entities within the activation unit by executing the appropriate administrative operations. Typically, it first issues an instantiation operation for all the entities in the activation unit and then unlocks them to put them back into service. The activation unit must list all the entities on which these operations need to be issued. For AMF entities, these operations are the unlock-instantiation and the unlock administrative operations respectively.

To guarantee controlled activation, entities newly added by the upgrade step must be added in a deactivated state. For AMF entities with administrative state, this means

that the initial administrative state needs to be locked, which needs to be taken into account at the specification of the appropriate IMM create operation.

The activation unit may be left empty if the campaign only intends to remove certain entities from the configuration.

Finally, the activation unit is equivalent to the deactivation unit when only existing entities are upgraded, no entities are removed from the system, or added to it. Such an activation unit is termed a **symmetric activation unit**.

If the upgrade of entities in a symmetric activation unit requires neither offline uninstallation nor offline installation (and therefore no lock and unlock operations), the campaign specification may merge the termination and the instantiation operations into a restart operation, if it is available for the symmetric activation unit.

3.3.2.3 Actions of the Upgrade Step

The current document defines the following ordered set of standard actions for an upgrade step.

1. Online installation of new software
2. **Lock deactivation unit**
3. **Terminate deactivation unit**
4. **Offline uninstallation of old software**
5. **Modify information model and set maintenance status**
6. **Offline installation of new software**
7. **Instantiate activation unit**
8. **Unlock activation unit**
9. Online uninstallation of old software

Alternatively, for restartable entities, a reduced set of actions is defined:

1. Online installation of new software
2. **Modify information model and set maintenance status**
3. **Restart symmetric activation unit**
4. Online uninstallation of old software

To maintain availability, the actions set in **bold** must be performed within the upgrade step. Note that the activation unit or the deactivation unit may be empty.

The old and new software are the sets of software bundles that need to be uninstalled and installed respectively. For each bundle, it is identified on which nodes these operations need to be performed.

The online installation of the new software (first action) may be performed in advance, in which case the first action—if performed within the step—acts as a verification of that installation.

The online uninstallation may be delayed for as long as required. To shorten recovery time in case of failure, it may be desirable to postpone the online uninstallation until the entire upgrade campaign has completed successfully.

3.3.3 Upgrade Procedure

An **upgrade procedure** applies the same upgrade step over a set of typically identical deactivation-activation unit pairs according to some constraints defined by an upgrade method.

3.3.3.1 Upgrade Scope

The composite set of deactivation units defines the **deactivation scope** of the upgrade procedure. The composite set of activation units defines the **activation scope**. The **upgrade scope** is the union of the activation and the deactivation scopes. The scope of symmetric activation units that belong to the same AMF parent entity, such as a service group or the cluster, is referred to as **symmetric upgrade scope**.

Entities within the upgrade scope usually have tighter dependency than those outside of the scope. For example, entities in the upgrade scope participate in the same redundancy schema and protect the same services.

3.3.3.2 Upgrade Method

The dependency among the software entities within the upgrade scope and the targeted service availability together determine the upgrade method appropriate for the upgrade procedure.

The **upgrade method** defines the constraints that must be observed when the upgrade steps are combined into a procedure; these constraints can be, for instance, the number of iterations, if any, and the ordering in which the upgrade steps are carried out. A particular upgrade method may define different sets of actions for its different upgrade steps.

This document defines the following upgrade methods:

- rolling upgrade and
- single-step upgrade.

3.3.3.2.1 Rolling Upgrade

The **rolling upgrade** iterates the same upgrade step (the same set of actions as defined in Section 3.3.2.3) over a set of similar deactivation-activation unit pairs one by one until the entire upgrade scope is covered.

The main advantage of the rolling upgrade is that it takes out of service one deactivation unit at a time for upgrade while the rest of the system is providing the service.

By selecting the deactivation and activation units in accordance with the redundancy model, the rolling upgrade ensures that there will be no service outage, or it will be limited to the outage caused by a single deactivation unit during the execution of the upgrade procedure (provided that failures do not reduce the number of redundant entities).

The limitation of this method is that entities of different versions of the base entity type coexist within the system without any restrictions; therefore, they must be able to collaborate for high-availability purposes. For example, they may need to act as peer entities within the same redundancy schema protecting the same service, which may require them to exchange state information.

Provided that the required availability of services can be maintained, some optimization of a rolling upgrade procedure may be possible by executing multiple upgrade steps concurrently. However, by default, such an optimization is not permitted. If such optimization is desirable, it needs to be enabled in the upgrade campaign specification by setting the appropriate attribute of each upgrade procedure for which it is applicable. The implementation of this optimization feature is optional.

3.3.3.2.2 Single-Step Upgrade

A **single-step upgrade** means that the upgrade scope is not divided into multiple deactivation activation unit pairs. There is only one such pair. The upgrade step is applied to all software entities in the upgrade scope simultaneously by taking out of service the entire deactivation unit at the beginning of the step and then re-activating the entire activation unit after the completion of the upgrade step.

Since the upgrade is carried out simultaneously on all software entities of the upgrade scope, none of them can provide service during this operation. This causes service

outage for all service instances that rely exclusively on the entities involved in such an upgrade.

The advantage or the reason why this method is still applied is that it eliminates the compatibility issues among the entities of the upgrade scope. This is also the simplest upgrade method to implement. It is appropriate for entities that are not providing services any more and need to be removed from the deployment configuration or for new entities that need to be added to the deployment configuration.

In certain cases—after a failure—single-step upgrade may be the only applicable method.

3.3.3.3 Procedure Execution Level

Dependencies between scopes, such as, for example, compatibility requirements, may impose a certain ordering of the upgrade procedures. To facilitate such ordering, each upgrade procedure has an assigned **execution level**. Procedures of the same execution level may be executed in parallel. A procedure with a higher execution level cannot be started before all procedures of the lower execution level have been successfully completed.

Thus, an upgrade campaign can be defined as an ordered set of upgrade procedures. The scope of any of these procedures may have dependency only on scopes of procedures that are upgraded earlier in the same campaign.

3.3.4 Service Outage

From an availability perspective, the most important concern of upgrades is whether all the services of the cluster can be provided or not during the campaign.

To protect against failures, AMF applications deploy redundancy, which may also be used during upgrade operations by upgrading these redundant entities in a sequence (for example, as described in Section 3.3.3.2.1), so there is always some entities that provide the services. Obviously, deactivating some software entities during the upgrade may reduce the capacity of the system; however, the services can still be provided. Such a reduction in the capacity is referred to as **service degradation**.

It may also happen that some of the services cannot be provided at all during the upgrade; this interruption of services is referred to as **service outage**. Software entities that are not deployed in redundancy cannot be upgraded without service outage.

With respect to service outage, this document considers only AMF entities. Therefore, the service outage is defined as the list of service instances that cannot be provided during the upgrade (that is, their assignment state becomes unassigned).

For each procedure of the upgrade campaign, a **minimum service outage** can be calculated by matching its deactivation units against the deployment configuration in which all service instances are fully-assigned and there are no disabled entities. This is the service outage that is unavoidable for the given upgrade procedure. Based on this, an **acceptable service outage** is defined for each upgrade procedure. The acceptable service outage must be at least the minimum service outage to be able to satisfy the campaign prerequisites at least under ideal conditions. The acceptable service outage may allow for further unassigned service instances.

The difference between the minimum service outage and the acceptable service outage signifies the importance or the urgency of an upgrade campaign. Ultimately, an upgrade may allow for all service instances to be unassigned. Such an essential upgrade could be one that increases system security or fixes bugs, and so prevents further service degradations and outages.

The acceptable service outage attribute of an upgrade procedure is used before initiating an upgrade campaign: Similarly, as at the calculation of the minimum service outage, the deactivation units are matched against the actual deployment configuration with the actual assignment state of the service instances at the time the upgrade campaign is about to start. This gives the **expected runtime outage** that must not exceed the acceptable service outage of the procedure. If it does, the campaign cannot start.

Note that once the campaign has started, as it unfolds, the service outage may exceed the acceptable service outage, but this fact will not terminate the campaign.

3.4 Upgrade Periods

3.4.1 Upgrade Procedure Period

For each upgrade procedure, a time estimate is defined within which the procedure is expected to complete. It is called the **upgrade procedure period** (τ). This time estimate should allow for the completion of the upgrade procedure itself (τ_p) and for the verification of the upgrade scope (τ_v). Accordingly, the upgrade procedure period is

$$\tau = \tau_p + \tau_v . \quad \textbf{(EQ 1)}$$

The upgrade procedure period helps to estimate the length of the expected service outage caused by the procedure; it is also used to estimate the upgrade campaign period.

3.4.2 Upgrade Campaign Period

For each upgrade campaign, and based on the upgrade procedure period and the sequencing of the procedures, a time estimate can be calculated within which the campaign is expected to complete. It is called the **upgrade campaign period** (*T*).

This time estimate consists of the addition of three time durations:

- Time for the completion of the upgrade campaign itself (T_c). The time necessary for the completion of the upgrade campaign depends on the periods of the individual upgrade procedures and their sequencing. The upper limit is:

$$T_c = \sum_{i=0}^{n} \tau_i \ , \qquad \textbf{(EQ 2)}$$

  where *n* is the number of procedures in the campaign and τ_i is the period of the *i*th procedure.

- Time for the verification of the system (T_v) after the completion of the campaign. Note that the verification of the individual scopes is included in the upgrade procedure period. This is an additional time that shall allow cross-system verification and that may include even an observation period.
- Slack time (T_s)—optional buffer period.

Accordingly, the upgrade campaign period is

$$T = T_c + T_v + T_s. \qquad \textbf{(EQ 3)}$$

The upgrade campaign period is used

- to schedule a proper maintenance window ($T<T_w$) for the planned upgrade campaign,
- in comparison with the elapsed time to estimate the time remaining until the completion of the upgrade campaign, and
- in case of failure to determine whether a rollback or a fallback is more appropriate.

Since the rollback procedure is typically symmetric to the upgrade campaign itself, it is assumed that it will take as much time to roll the system back from its current con-

figuration to the one at the beginning of the campaign as it took for the campaign to reach the current configuration. At the end of the rollback, the system also needs to be verified; therefore, the time needed for verification must be accounted for. This means that if the failure occurs at the moment *t* (elapsed time) of the upgrade campaign, it is expected that it will take $t+T_v$ to rollback the campaign and verify the rollback. Thus, the duration of the entire operation will be *2* x *t* + *Tv*. If the maintenance window does not permit this duration, system fallback could be considered a faster way to bring the system back into an operational state[1].

3.5 Upgrade-Aware Entities

For applications that require application-level actions associated with an upgrade campaign, an API is exported by the Software Management Framework: the Software Management Framework API allows client processes to register their interest in being informed about upgrade campaigns initiated by the Software Management Framework and about their progress. The Software Management Framework informs registered client processes about the progress by invoking the appropriate callback; these registered processes can in turn synchronize application-level actions with the upgrade campaign. The registered processes report the result of these application-level actions with a response. The Software Management Framework uses these responses in its decisions about the course and the outcome of the upgrade campaign.

This API interface is described in detail in Chapter 8; it is expected to be used by management entities within an application that are capable of interpreting such callbacks and coordinate application-level actions accordingly. These entities are referred to as **upgrade-aware entities**.

If an upgrade-aware entity initializes the Software Management Framework interface while an upgrade is already in progress, and one of its process registers a callback with the Framework, the Framework only invokes subsequent callbacks.

3.6 Typical Software Management Information Flow

FIGURE 4 shows the typical flow of information for software upgrades.

When a software vendor provides a new software bundle, an entity types file describing the content of the software bundle shall be supplied with the bundle. The format of the entity types file must follow the XML schema described in Chapter 7.

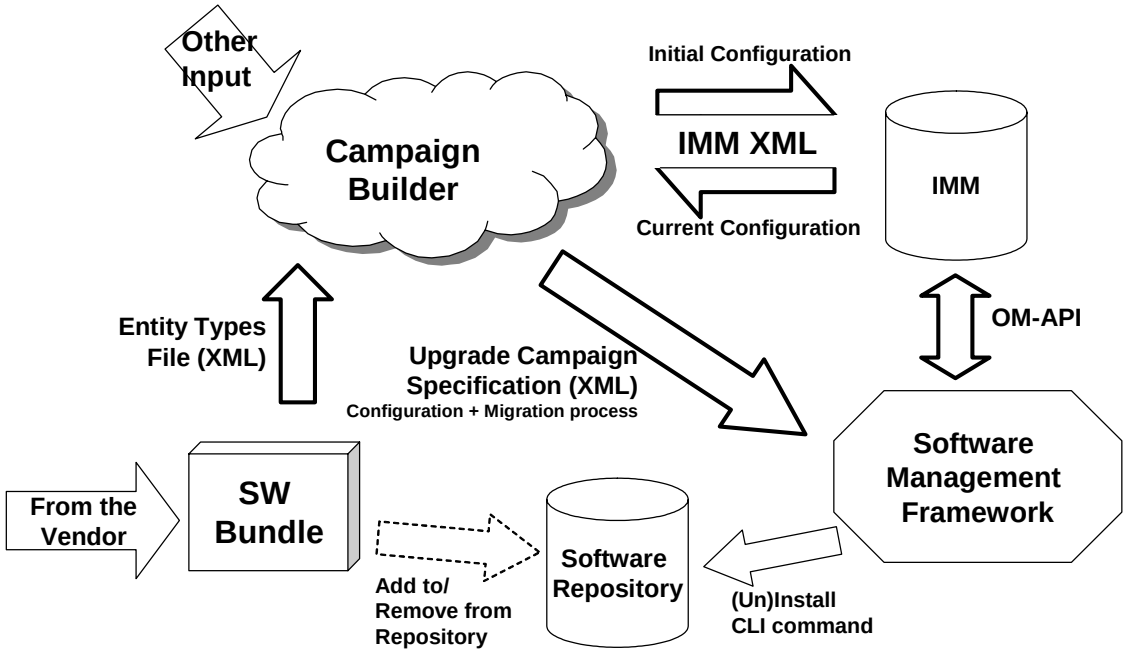
1. The fallback time is typically expected to be similar to the system startup time.

This entity types file is expected to be processed by some tool—referred to in the figure as **Campaign Builder**—which is used by the site designer or integrator to generate an upgrade campaign specification.

Besides the entity types file, the Campaign Builder will need additional input:

- Information about the current configuration of the SA Forum system to be upgraded. This configuration shall be obtained from the Information Model Management Service. The format of this information is standardized by the IMM specification [4].
- Other input specific to the site and to the Campaign Builder tool, such as performance expectations, non-SA Forum-related application characteristics, and so on. This portion of the input information is not standardized by the SA Forum in any way.

FIGURE 4 Typical Software Management Information Flow for an Upgrade



For a running system, the Campaign Builder shall generate an upgrade campaign specification file in XML according to the XML schema provided in Chapter 6. Such an upgrade campaign specification file contains information about the new configuration of the SA Forum system and the process that shall be followed to migrate the cluster to this new configuration.

The upgrade campaign specification file is provided as an input to the Software Management Framework. To be able to execute the upgrade campaign, the software bundle must also be delivered to the system's software repository. This delivery is performed in an implementation-specific way, as image management is out of the scope of the current release.

Once the software bundle has been delivered to the software repository, and the upgrade campaign specification has been provided to the Software Management Framework, the administrator may initiate the execution of the upgrade campaign using the SMF administrative API.

As part of the initialization of the upgrade campaign, the Software Management Framework must update the software catalog based on the information provided in the campaign specification. This means that it creates or verifies the software bundle and all the entity type objects in the information model. Currently, it is not specified how the presence of a software bundle in the software repository is verified against the software bundle information model object.

During the execution of the campaign, the Software Management Framework will update the upgrade campaign model in IMM to reflect the status of the execution. It will also interact with the Availability Management Framework and other SA Forum Services as specified in the upgrade campaign specification. Particularly, it will use the installation and uninstallation CLI commands specified by the software bundle object to install the software bundle to target locations and to remove the software bundle from target locations.

FIGURE 4 also shows that such a Campaign Builder tool can be used to generate an initial configuration for an SA Forum system. However, in this case, the initial configuration is specified in the format required by the Information Model Management Service, which shall use it to create the internal representation of the system's initial information model. SA Forum Services must obtain their respective configuration from the Information Model Management Service directly in this scenario. This scenario also assumes that all the necessary software bundles are available in the software repository and have been installed on the target locations within the cluster.

4 Failure Detection and Failure Handling

4.1 Failure Detection

As an upgrade campaign unfolds, its actions need to be verified continuously, so that, if an error is detected, corrective actions may take place as soon as possible.

The Software Management Framework verifies initially whether it is feasible to initiate an upgrade campaign at the given moment in the given state of the system, that is, the campaign prerequisites are checked.

Subsequent verifications are done as follows.

- The Software Management Framework must evaluate the result of each individual action performed, such as the success or failure of a CLI operation during an upgrade step.
- The Software Management Framework must evaluate the results reported back by **registered processes** of upgrade-aware entities. These are processes that register with the Software Management Framework to be called back during the upgrade campaign under certain circumstances (see Section 6.1.1.4).
- The Software Management Framework must monitor the operational state change notifications generated by the Availability Management Framework that indicate that the disabled entity was involved in a maintenance campaign (see Section 3.3.2.2).
- The Software Management Framework may monitor other notifications and alarms in the system to correlate them with the actions performed by the campaign.

Prerequisites are checked only once at the initiation of the upgrade campaign. All other type of verifications are performed regardless of whether the upgrade campaign is in its forward (upgrade) path or it is in a rollback (recovery) path.

The following subsections mainly contains recommendations on the inputs that shall be used for verification. The exact decision mechanism using these and possibly other inputs is implementation-specific.

4.1.1 Upgrade Prerequisites

Before an upgrade campaign starts, the following prerequisites must be checked at a minimum.

1. The Software Management Framework is operational.
2. The software repository is accessible.
3. There is no other upgrade campaign in progress.
4. The currently running version of the software is available in the software repository.
5. The specifics of the upgrade campaign have been provided, and the campaign is still applicable.
6. The desired version of the software is available in the software repository, and all the dependencies of the required packages have been checked and are satisfied.
7. All affected nodes are able to provide the resources (for instance, sufficient disk space and proper access rights) needed to perform the upgrade campaign.
8. The Software Management Framework is able to obtain the administrative ownership for all necessary information model objects.
9. The target system is in a state such that the expected service outage does not exceed the acceptable service outage defined for the campaign.
10. Upgrade-aware entities are ready for an upgrade campaign.
11. Any necessary backup is created.

If any of these checks fails, the upgrade campaign must not start. If the upgrade campaign has been initiated, it must terminate immediately. Note that after the correction of the failed prerequisite, the campaign may be re-attempted.

Prerequisites 1. and 2.

Prerequisites 1. and 2. are verified in an implementation-specific way.

Prerequisite 3.

Prerequisite 3. is verified based on the Software Management Framework Information Model by checking that no upgrade campaign object shows the campaign in a state that is not an initial or a final state according to the state model (see Section 5.3).

Prerequisite 4.

Prerequisite 4. is partially verified based on the Software Management Framework Information Model by checking that all the configured entities refer to versioned entity types that have their software bundle objects in the software catalog. The actual images associated with the software bundles are checked in an implementation-specific way.

Prerequisite 5.

An upgrade campaign is specified in the upgrade campaign specification file (see Chapter 6) provided to the Software Management Framework. The Software Management Framework in turn checks the correctness of the XML file and whether the campaign is still applicable. An upgrade campaign is applicable as long as the time stamp referring to the deployment configuration based on which this upgrade campaign was created is not older than the time stamp associated with the current deployment configuration (see Section 6.2.2). If no time stamp is given in the campaign specification, the campaign specification is applicable at any time.

If the verification of this prerequisite is successful, the Software Management Framework is allowed to create the upgrade campaign model based on the provided upgrade campaign specification file.

Note that the verification of this prerequisite may need to be performed in several stages to take into consideration that (a) the administrative action of adding the upgrade campaign object to the information model changes the time stamp associated with the current deployment configuration and that (b) there might be a time gap between the addition of the upgrade campaign object to the information model and the actual execution of the upgrade campaign represented by the upgrade campaign object.

Prerequisite 6.

The upgrade campaign specification file indicates the software bundles that are necessary for the execution of the campaign. The Software Management Framework must verify that these bundles are already in the software catalog of the Software Management Framework Information Model. The actual images associated with the software bundles are checked in an implementation-specific way.

If the software bundles are not in the catalog yet, the Software Management Framework shall check in an implementation-specific manner that these bundles have been added to the software repository, and, if this is the case, the Software Management Framework adds the corresponding objects to the software catalog.

Prerequisite 7.

Prerequisite 7. shall be verified in an implementation-specific way.

Prerequisite 8.

As part of the verification of prerequisite **8.**, the Software Management Framework shall obtain (using the IMM OM-API) the administrative ownership for at least the information model objects that represent entities

- listed in any of the deactivation or symmetric activation units of any upgrade procedures within the upgrade campaign and
- targeted by any actions of the upgrade campaign, namely, by actions specifying administrative operations or information model changes.

Prerequisite 9.

Prerequisite 9. is verified by projecting the deactivation units of the different procedures onto the current state of the AMF cluster and by determining whether they would cause a service outage that is still acceptable, as indicated for each procedure in the upgrade campaign specification (see also Section 3.3.4 and Section 6.4.1). This document does not define any further how this comparison is carried out.

Prerequisite 10.

Prerequisite 10. is verified using callbacks to registered processes of upgrade-aware entities. When such a process receives a callback at campaign initiation, it shall respond whether it is ready for the start of an upgrade campaign. This means that the registered process is currently not involved in any activity that may be jeopardized by an upgrade campaign (for example, its activity could be affected by the reduced redundancy during the upgrade), or that it would not jeopardize the success of the upgrade campaign.

The upgrade campaign specification explicitly distinguishes such callbacks (see Section 6.3.3.2.1) from other custom callbacks. It is up to the upgrade campaign developer to determine the need of this verification. The registered process must be able to evaluate the conditions under which the upgrade campaign shall be rejected and communicate the result to the Software Management Framework.

Prerequisite 11.

Prerequisite 11. requires the successful creation of a backup at system-level and, if necessary, at application-level.
The creation of the system backup is implementation-specific; however, it needs to be aligned with requirements stated in Section 4.2.1.4.

The Software Management Framework requests the creation of the application-level backup by invoking the appropriate callbacks of registered processes. The callback parameters are defined in the upgrade campaign specification (see Section 6.3.3.2.2). The registered processes shall indicate in a response the result of the operation. These responses and the success of the system backup are evaluated to determine if the campaign may start.

4.1.2 Upgrade Step Verification

During an upgrade campaign, two types of actions are carried out.

- Actions that return their result immediately (for instance, CLI commands for the installation), and this result is directly applicable to the upgrade step within which it is performed (see Section 4.1.5). The Software Management Framework can use this return value immediately to decide about the outcome of the upgrade step.
- Actions that do not return their result immediately. The Software Management Framework needs to rely on the error reports propagated by the Notification Service. These notifications and alarms may occur with some delay; therefore, the failures they report are referred to as asynchronous failures and are handled at the campaign level, as described in Section 4.2.1.3.

Application-level verification is performed by the registered processes of upgrade-aware entities. Upon a callback, these processes perform the actions identified by the callback parameters (see Section 6.1.1.4), and they shall report in a response to the Software Management Framework any failure they detect. The Software Management Framework uses these responses to determine how the campaign shall proceed (see Section 8.5.2). Note that a failure detected at the application level may be caused by an entity other than those affected by the current upgrade step; however, if a failure is reported in response to a callback during the step, the Software Management Framework interprets it as a failure of the upgrade step. Also, if a registered process fails to respond within the time limit specified for the callback in the campaign specification, this timeout does not cause the upgrade step or the upgrade campaign to fail. This approach was taken because the upgrade-awareness is not a basic requirement for software upgrades. The timeout value is given merely to provide some time for application level processes to perform their actions and respond, before the upgrade step is considered completed by the Software Management Framework.

The upgrade step is considered to be completed successfully after all actions are completed successfully, and the Software Management Framework has received the acknowledgement of the success from the registered processes of upgrade-aware entities that were called back in the given step, or the callbacks timed out.

If any part of the step verification fails, the failure handling takes over as described in Section 4.2.2.

4.1.3 Upgrade Procedure Verification

Once the upgrade procedure is completed, its entire scope needs to be verified.

To wrap up a procedure, the upgrade campaign specification described in Chapter 6 allows for the specification of verification actions at the end of each procedure in accordance with Section 6.1.1. The procedure verification may include the execution of CLI commands, administrative operations, IMM operations, application-level verification using the upgrade API, or a combination of these actions.

If any part of the procedure verification fails, the upgrade failure handling takes over as described in Section 4.2.2.

It is recommended that upgrade procedures are designed so that any potential failure will manifest as soon as possible in the system. For example, in case of a rolling upgrade in two steps, it is desirable that the upgraded entities of the first step receive an active assignment before the entities of the second step are upgraded. This way, if an upgraded entity of the first step fails, only entities of the first activation unit need to be repaired by rolling back. Entities of the second activation unit remain intact and can provide service.

4.1.4 Upgrade Campaign Verification

Once the last upgrade procedure is completed, the Software Management Framework must verify that the entire system is operational and no further actions are necessary. Since the verification of the individual upgrade procedures cannot guarantee that all the upgraded entities are fully capable of operating and providing their services integrated into their runtime environment, this verification action is essential for the proper completion of the upgrade campaign. This system-wide verification is necessary even if the campaign failed and was rolled back.

As part of the campaign verification, the upgrade campaign specification (see Chapter 6) allows for the specification of verification actions. These actions may include the execution of CLI commands, administrative operations, IMM operations, application-level verification using the upgrade API, or a combination of these actions. These verification actions must be performed before the campaign may be committed (see Section 6.5.1).

Due to the delay in the manifestation of some errors, an observation period may also be specified. If specified, the upgrade campaign may not be committed before the expiration of this timer.

After all the verification actions have succeeded, the timer for the observation period has expired (if specified), and no error has been detected, the campaign may be committed. The success of the upgrade campaign or its rollback must always be confirmed by an administrator by issuing the commit administrative operation (see Section 9.3.2).

If the upgrade campaign has been committed, the Software Management Framework proceeds with the remaining wrap-up actions specified for the upgrade campaign (see Section 6.5.2). The results of these actions do not impact the outcome of the campaign any more. These actions may include execution of CLI commands, administrative operations, IMM operations, application-level verification using the upgrade API, or a combination of these actions. As part of the wrap-up actions, the Software Management Framework must reset the maintenance status of all entities for which it has been set during the campaign.

The upgrade campaign specification may also specify a second timer that ensures that an administrator triggered system fallback operation (see Section 4.2.2.4) can still recover the system state stored in the backup at the beginning of the campaign.

If the upgrade is deemed to have failed, the Software Management Framework commences with the recovery action ordered by the administrator as described in Section 4.2.2.

4.1.5 Exit Status

The valid range for the exit status for any CLI command initiated by the Software Management Framework is

0 <= exit status <= 255.

CLIs have a zero exit status in case of success, non-zero in case of failure. Values in the range

200 <= exit status <= 254

have either predefined meanings or are reserved for future usage. The reaction of the Software Management Framework to these errors is described as necessary in the next sections.

4.2 Failure Handling

During the upgrade campaign, the Software Management Framework deploys some protective measures to facilitate recovery operations as part of the upgrade failure handling. This section first presents the protective measures taken before and during

an upgrade campaign and then describes how they are used in the different recovery operations to handle different failures.

4.2.1 Protective Measures

The purpose of the different protective measures is to minimize losses due to failures during the upgrade campaign and to speed up the system's recovery. The mandatory measures that must be implemented by a Software Management Framework implementation are:

- the backup and
- the upgrade history.

In addition, upgrade-aware entities may synchronize their following actions with the upgrade campaign:

- application log used together with application-level backup and
- application checkpoint.

Appropriately, the relevant callbacks need to be specified in the upgrade campaign specification for all interested applications, which shall register with the Software Management Framework to receive the callbacks at the specified stages.

4.2.1.1 Backup

A system backup is a persisted image that can be used after the system was ordered a full cold restart to restore the system to the state in which it was at the moment the backup was created. Depending on the implementation, this requirement introduces further requirements with respect to the content of the saved image.

This document only defines the requirements from the perspective of the Information Model Management Service and the Availability Management Framework.

The Information Model Management Service stores the cluster configuration and all the persistent state information; therefore, it is mandated that a copy of the IMM contents is created and included in the system backup. This copy shall cover all the needs of the Availability Management Framework to restore the state of the entities it manages. Note, however, that a particularly AMF component has an associated software installation that the Availability Management Framework uses to control the life cycle of the component. Since image management is not in the scope of the current release of Software Management Framework specification, it is left to the implementation to resolve the issue of restoring the appropriate software images that have potentially changed during a (failed) campaign.

In addition to the Availability Management Framework and Information Model Management Service, applications may have their own data that need to be saved, so that the state of the application can be restored. Only the application itself is aware of such a need; therefore, the Software Management Framework shall invoke a callback to the registered processes of the application to inform them about the necessity to create such a backup (see Section 6.3.3.2.2). It is recommended that an application that requires the synchronization of its backup with an upgrade campaign is upgrade-aware, and that the upgrade campaign specification specifies the necessary callback.

The backup needs to be created when the system is stable and consistent, that is, its state is suitable to be restored in case of a fatal failure. As discussed above, the backup must contain all the necessary information to restore this stable state even after a cold restart. The backup image must be stored in a way that neither a system failure nor a cold start can affect it. On the other hand, it should be easily accessible during recovery to minimize the recovery time.

The system backup is created as part of the prerequisite checks when an upgrade campaign is initiated. The Software Management Framework is responsible for initiating the backup at system-level and triggering the initiation of the application-level backup through callbacks. If the backup fails, the upgrade campaign must be terminated immediately.

The backup is required for the fallback recovery action; therefore, its implementation is mandatory.

4.2.1.2 Upgrade History

During the upgrade, the Software Management Framework maintains a history of the executed actions. This upgrade history needs to contain enough information to allow for

1. undoing the executed actions of the current upgrade step of a procedure, and for
2. rolling back the upgrade campaign.

The executed actions need to be undone in reverse order to restore the deployment configuration to the state that was in effect at the beginning of the upgrade step. This restoration is expected to be possible from the current running state even after a failure. Section 4.2.2.1 describes how this information is used.

For rolling back, it is necessary to remember the current configuration information of the entities targeted by the upgrade, as it represents the configuration information to which the system needs to return during a rollback. Section 4.2.2.3 describes the rollback operation.

The implementation of the upgrade history is mandatory, as it is required for the roll-back recovery operation. It is also necessary to undo any partially completed upgrade step. This document does not specify how this feature is implemented.

4.2.1.3 Detection of Asynchronous Failures of AMF Entities

For some software entities, an attribute indicating their maintenance status is specified. In particular, the Availability Management Framework defines the `saAmfSUMaintenanceCampaign` configuration attribute for service units. This attribute must be used by the Software Management Framework as follows.

The Software Management Framework must determine all the software entities that are partially or fully upgraded by an upgrade step and set their maintenance status to contain the current campaign name, after the IMM modification actions have been performed. For AMF entities, this means that if a service unit is upgraded by an upgrade step, its `saAmfSUMaintenanceCampaign` attribute must be set by the Software Management Framework before the activation unit is activated. If a component is upgraded by an upgrade step, the attribute of the enclosing service unit must be set as if the service unit itself was upgraded.

This setting blocks any auto-repair performed on failed entities by the object implementer, in particular by the Availability Management Framework. Instead of the auto-repair, the failed entity, such as a service unit, is disabled right away when its failure is detected, and an operational state change notification is generated by the object implementer. This operational state change notification indicates as additional information an `infoId = SA_AMF_MAINTENANCE_CAMPAIGN_DN` and its `infoValue` is the DN of the upgrade campaign.

The Software Management Framework must subscribe to operational state change notifications generated by object implementers, particularly by the Availability Management Framework; and must monitor them if an ongoing campaign is indicated as the reason for disabling. If such an operational state change notification is received, the Software Management Framework must interpret it as a potential failure of the upgrade campaign and suspend the campaign execution according to Section 5.3, Section 9.3.1, and Section 9.3.4.

It is the administrator's responsibility to resolve the situation and decide whether the campaign is continued, rolled back, or a fallback is performed. The administrator also has to take all the measures appropriate to this choice. The Software Management Framework must facilitate the administrator's task by indicating the failed service unit that triggered the campaign suspension in the `saSmfCmpgError` attribute of the upgrade campaign object (see Section 3.3.1.1.1). It must also collect similar subse-

quent service unit failures and update the `saSmfCmpgError` attribute until the administrator issues further administrative operations. For example, if the administrator decides that the failure is not the consequence of the upgrade, and the campaign should continue, the administrator may decide to repair the failed service units to make them available again, as the Availability Management Framework is blocked to perform such a repair.

When the Software Management Framework receives an execute (Section 9.3.1) or rollback (Section 9.3.4) administrative operation, it clears the `saSmfCmpgError` attribute of the upgrade campaign object and assumes that the administrator was aware of all the failures that were detected up until that moment and therefore has taken care of them. In other words, the Software Management Framework is not expected to perform any kind of checking, verification, or repair.

When an upgrade campaign is committed, the Software Management Framework must reset all the maintenance status attributes that refer to the campaign being committed. Beyond this point, it cannot determine whether a failed entity was upgraded by the campaign or not.

4.2.1.4 Handling Persistent Changes During Upgrade

An SA Forum system continues to provide its services during an upgrade campaign, which means that some of the information saved in the backup image may become obsolete due to changes occurring as results of the normal operation and administrative actions. Thus, if the backup is restored in a fallback operation these changes are lost.

Transaction-based applications and systems use logs to recover such changes. In such a log, they record any change with respect to the saved backup image. If the backup image needs to be restored, after its restoration, these logged changes are re-applied to recover the lost state. This procedure is referred to as **rollforward**.

Persistent changes that occur while a system is upgraded fall into two categories:

- those that are the result of the upgrade campaign itself and
- changes due to the normal operation.

4.2.1.4.1 Changes Caused by the Upgrade

The intention of the restoration of a backed up image is to get rid of the changes that were introduced by the upgrade, as they might have led to an inconsistent system state. Therefore, these changes must not be logged.

The information model maintained by the Information Model Management Service is part of the backup image. The model is manipulated by the upgrade campaign, and these changes must not be restored after a fallback. Therefore, this specification does not require that the Software Management Framework or the Information Model Management Service maintain any logs during an upgrade.

This also means that during the upgrade campaign any persistent change to the information model that is not part of the upgrade —even if it is caused by normal operation or administrative actions—may be lost after a fallback, as far as the Software Management Framework or the Information Management Service are concerned (see also the next section).

4.2.1.4.2 Changes Caused by Normal Operation

Changes due to normal operation that have to be recovered after a fallback need to be recorded in a log, so they can then be used together with the backup image.

The current document does not identify any SA Forum-specific data that need to be restored in addition to the backup image and, therefore, would require logging. Applications, however, may have such data.

Since the Software Management Framework is not aware of any persisted data changes at the application-level, any such log needs to be maintained by the application itself. It is also the responsibility of the application to apply these logs automatically after a fallback, when the application state is restored from the backup.

If the application level backup and logging needs to be synchronized with the progress of an upgrade campaign, the application shall utilize the Software Management Framework API, and the upgrade campaign needs to specify the relevant callbacks. The Software Management Framework is not required to signal the rollforward action to upgrade-aware entities.

4.2.2 Recovery Operations

When an action within an upgrade step fails, and the Software Management Framework can detect this failure immediately, the first reaction of the Software Management Framework is to undo the effects of the already completed actions of this upgrade step to bring the system state back into a known state, namely the state at the beginning of the upgrade step. This recovery is done automatically. Depending on the success of this recovery operation, the impact of the failure is contained at the step level or needs to be widened to the campaign.

If a failure is detected with some delay (e.g. asynchronously through notifications as described in Section 4.1.2), or at a procedure or campaign verification, the failure affects the entire campaign.

Accordingly, different recovery operations become applicable to the campaign.

In case the failure is contained at step level

- retry of the upgrade step.

In case the failure affects the campaign

- rollback of the campaign or
- system fallback

In case of a failure during rollback

- system fallback

Only the retry of an upgrade step is executed automatically by the Software Management Framework (details are in Section 4.2.2.2). All the other recovery operations need to be selected or confirmed by the administrator explicitly before the Software Management Framework embarks on their execution.

4.2.2.1 Undoing an Upgrade Step

Section 3.3.2.3 presents the standard actions of an upgrade step. The Software Management Framework may encounter an error during the execution of any of these actions. To undo such a failed step, the effects of its already executed actions need to be reversed. This is done using the upgrade history maintained by the Software Management Framework.

The actions recorded in the history and which belong to the failed step are undone one by one in a reverse order by executing the reversing actions of the recorded action. The reversing actions for each of the actions of the standard upgrade step are presented in the following table.

Note that even if the old software was not completely uninstalled during the failed step, it is necessary to execute the installation operation to verify the installation. If the offline uninstallation action was not empty, its reversing action should fulfill this need. However, if the offline uninstallation action was empty, only the online installa-

tion operation can provide this functionality even if the online uninstallation was not executed yet.

Table 1 Reversing Actions Depending on the Upgrade Step

Upgrade Step Action	Reversing Action
Online installation of new software	Online uninstallation of new software
Lock deactivation unit	Unlock deactivation unit
Terminate deactivation unit	Instantiate deactivation unit
Offline uninstallation of old software	Offline installation of old software
Modify information model and set maintenance status for activation unit	Reverse information model modifications and set maintenance status for deactivation unit
Offline installation of new software	Offline uninstallation of new software
Instantiate activation unit	Terminate activation unit
Unlock activation unit	Lock activation unit
Restart activation unit	Restart activation unit
Online uninstallation of old software	Online installation of old software

If undoing the step succeeds, and the retry of the failed step is permitted, the re-execution of the step is initiated automatically by the Software Management Framework (see Section 4.2.2.2). In this case, the failure is considered to be contained.

If undoing the step succeeds, but a retry is not permitted, or the allowed number of retry attempts has been exceeded, the impact of the failure is widened to the upgrade procedure and to the entire campaign. An administrative intervention is required to proceed (see Section 5.1.6).

The choices available for the administrator are:

1. correct the error that caused the failure of the step and force a retry of the failed step (see Section 4.2.2.2),
2. initiate a rollback (see Section 4.2.2.3), or
3. initiate a system fallback (see Section 4.2.2.4).

The first option should cover error situations for which the problem can be easily fixed by the administrator on the spot, for example, if it turns out during software installation that there is not enough disk space at the target location. Typically, these are errors

that can be detected immediately during the execution of the upgrade step (see Section 4.1.2).

If undoing the step also fails, the system state is considered to be unknown due to the double-failure. As a result, the administrator is left with no other choice than to initiate a system fallback. See also Section 5.1.5.

If rollback is selected by the administrator, the Software Management Framework shall inform the registered processes of upgrade-aware entities about the initiated recovery operation.

4.2.2.2 Retry of an Upgrade Step

Once the upgrade step is undone, and if the retry is permitted, the Software Management Framework automatically initiates the re-execution of the upgrade step.

The number of permitted retry attempts is configurable. With each retry attempt, a retry count is incremented, and once it exceeds the configured number of retry attempts, any subsequent failure of the given upgrade step results in widening the failure scope to the procedure and to the campaign, which will be suspended.

From the suspended state, an administrator may still force a retry attempt of the upgrade step using administrative operations. It is left to the discretion of the administrator to decide if this forced re-execution is appropriate and perform any necessary repair actions. See also Section 5.1.6, Section 5.2.3, Section 5.3.7, and Section 9.3.1.

4.2.2.3 Rollback

Rollback is the recovery operation that reverts the successfully completed steps of an upgrade campaign to recover the deployment configuration effective at the beginning of the upgrade campaign. It is applicable only at step boundaries. Rollback is a graceful recovery, as the system continues to provide its services during rollback, similarly as it does before during the campaign.

Nevertheless, rollback does not necessarily restore the system state that was in effect at the beginning of the campaign, as any changes that happened due to the system's normal operations generally remain intact. Some losses may occur, but they are limited at application-level typically to the restoration of the last checkpoint.

The administrator may initiate the rollback almost any time during the upgrade campaign after having suspended it (see Section 5.3.8). However, the rollback is usually initiated by an administrator after an upgrade step has been undone because either

the retry was not permitted, or the allowed number of attempts was exceeded (see Section 5.3.7).

The campaign may not be in a final failure state at the initiation of a rollback (see Section 5.3.13).

Rollback is applied to the entire campaign. Thus, all executing procedures of the campaign must be suspended at a step boundary first (see Section 5.3.7 and Section 5.3.8), and then the rollback may be applied (see Section 9.3.4).

The Software Management Framework informs registered processes of upgrade-aware entities that a rollback has been initiated. After the rollback has been signaled, upgrade-aware entities must interpret subsequent callbacks as part of the rollback operation.

At its completion, the outcome of the rollback must be verified the same way as it is verified for an upgrade (see Section 4.1.4).

The rollback operation uses the upgrade history as described in the following section.

4.2.2.3.1 Campaign Rollback

The campaign rollback is symmetrical to the upgrade campaign itself. It initiates the rollback of each procedure in the reverse order to their execution level, starting from the current execution level.

4.2.2.3.2 Procedure Rollback

The rollback of a given procedure is symmetric to its forward execution. During the procedure rollback, completed steps are rolled back one by one in reverse order.

4.2.2.3.3 Step Rollback

The step rollback is similar to its forward execution, except that it deactivates the entities in the original activation unit and reactivates those of the original deactivation unit. The configuration information of the former deactivation unit shall be available from the upgrade history.

Step rollback defines the following actions:

1. Online installation of old software
2. **Lock activation unit**
3. **Terminate activation unit**
4. **Offline uninstallation of new software**

5. **Modify information model to old configuration and set maintenance status for deactivation unit**
6. **Offline installation of old software**
7. **Instantiate deactivation unit**
8. **Unlock deactivation unit**
9. Online uninstallation of new software

Alternatively, the reduced set of actions for restartable entities is the following:

1. Online installation of old software
2. **Modify information model to old configuration and set maintenance status**
3. **Restart symmetric activation unit**
4. Online uninstallation of new software

The first action in both cases restores the installation of the old software if it was uninstalled after the completion of the upgrade step. If this is not the case, it verifies the installation.

The maintenance status needs to be set for entities that are being restored by the rolling back step.

The last action in both cases can be postponed indefinitely.

The actions set in **bold** must be executed as part of the step rollback to ensure availability.

4.2.2.3.4 Failure During Rollback

A failure of an action during the rollback is treated similarly to an error during the upgrade campaign, except that the possible recovery options are reduced.

If an action of a rolling back step fails, the Software Management Framework immediately tries to undo the already completed actions of this rolling back step by reversing each of the performed actions in reversed order (see Section 5.1.7).

Table 2 Reversing Action Depending on the Step Rollback Actions

Step Rollback Action	Reversing Action
Online installation of old software	Online uninstallation of old software
Lock activation unit	Unlock activation unit
Terminate activation unit	Instantiate activation unit
Offline uninstallation of new software	Offline installation of new software
Modify information model to old and set maintenance status for deactivation unit	Reverse information model modifications and set maintenance status
Offline installation of old software	Offline uninstallation of old software
Instantiate deactivation unit	Terminate deactivation unit
Unlock deactivation unit	Lock deactivation unit
Restart activation unit	Restart activation unit
Online uninstallation of new software	Online installation of new software

If the undoing of the rolling back step is successful, and retry is permitted, the step rollback is retried automatically (see Section 5.1.8).

If the undoing of the rolling back step fails, or if it is undone successfully, but no more retries are permitted (see Section 5.1.9), the failure is widened to the entire campaign, and the only allowed recovery option is the system fallback. Note that a forced retry is not permitted at rollback[1]. See also Section 5.2.7 and Section 5.3.13.

4.2.2.4 Fallback

A **fallback** typically means a complete (cold) system restart using the image saved during the system backup. In other words, after the restart, the system is restored to the state that was in effect at the time when the backup was created.

1. This decision was made to keep the state machines simple and was not due to technical considerations and limitations.

A fallback is initiated by an administrator. It is implementation-specific how the initiated fallback operation is performed. As many of its aspects are beyond the current scope of SA Forum specifications, this document provides only information that needs to be considered by an implementer of the operation. The operation itself must be provided by an implementation. An implementation needs to be able to restore the software images appropriate to the deployment configuration together with the deployment configuration that was saved in the system backup. During fallback, the system is not expected to provide services.

System fallback reverses all the changes that happened to the different entities during the upgrade campaign. Any configuration change (applied by using the Information Model Management Service) that happened during the upgrade is lost after a fallback operation (see also Section 4.2.1.4).

To recover data losses at application-level, an application may maintain its own log. It shall be able to use this upgrade log to execute a rollforward operation as part of the restart operation automatically without expecting a callback or other actions from the Software Management Framework (see also Section 4.2.1.4.2).

4.2.2.4.1 Rollforward

The rollforward operation recovers changes that have taken place after a backup was created by reapplying the logged changes to the system state restored from the backup. It is typically used in connection with the fallback operation to restore changes resulting from normal operation of the system during the upgrade that would have been lost due to a fallback.

The current document neither defines nor requires rollforward for any SA Forum Service.

5 State Models

An upgrade campaign is modeled by a set of communicating finite state machines (FSM). A state machine is maintained for each object in the upgrade campaign model (see Section 3.3.1), that is, for the upgrade campaign, for each upgrade procedure within the campaign, and for each step within each procedure. The state of the upgrade campaign and its constituent parts are defined by the states of these finite state machines, as described in this chapter.

There is one upgrade campaign FSM in the Software Management Framework Information Model for each upgrade campaign specified by an XML file. The execution of this upgrade campaign is controlled by administrative operations applied to the upgrade campaign object. These administrative operations are interpreted as input signals to the associated finite state machine. The campaign FSM receives these signals from the administrator, reacts by state transitions and by communicating the appropriate signals to the FSMs representing the upgrade procedures of the campaign. This way, the campaign FSM controls the execution of procedure FSMs, receives the results of their execution (which is also reflected in state changes as appropriate), and returns the results to the administrator.

In turn, each upgrade procedure FSM communicates with the FSMs of its upgrade steps in a similar manner to control them and receive their results.

The following notation is used in the subsequent state diagrams:

- A double circle denotes the initial state of the FSM, the FSM is created in this state.
- A fat lined circle represents a final state, no exit transition is defined from such a state.
- Arrows represent transitions between states.
- Each transition is labeled with an input signal that triggers the state transition and an output signal that is produced as a result of the transition, if applicable. The input and output signals are separated by a slash.
- If a signal is received from an FSM or sent to another FSM, the signal is tagged with that FSM such as `Cmpg` for the campaign FSM, `Proc` for the procedure, and `Step` for the step. Signals with no tags are consumed by the FSM that produced them (for instance, `Done` / `Proc:Completed`).
- Signals in capital letters represent administrative operations (for brevity, the `SA_SMF_ADMIN` prefix is omitted). An administrative operation may span several states (for instance, `SA_SMF_ADMIN_EXECUTE`); therefore, its return is indicated as an output signal with the prefix `RTN_<admin operation>`. Signals in angle

brackets (for instance, <fallback>) are currently not standardized, but expected to be available in an implementation.

5.1 Upgrade Step State Model

FIGURE 5 presents the state diagram of an upgrade step FSM.

5.1.1 Initial State

The step FSM is created in the Initial state. The execution of the step is initiated when the Proc:Execute signal is received from the finite state machine of the upgrade procedure to which the upgrade step belongs. This signal moves the step FSM into the Executing state.

5.1.2 Executing State

In the Executing state, the actions of the upgrade step are executed one by one as described in Section 3.3.2.3.

If no failure is detected during the execution, and all actions—including possible verifications—complete successfully (Done), the step FSM transitions to the Completed state while it signals Proc:Completed to the procedure FSM to which it belongs.

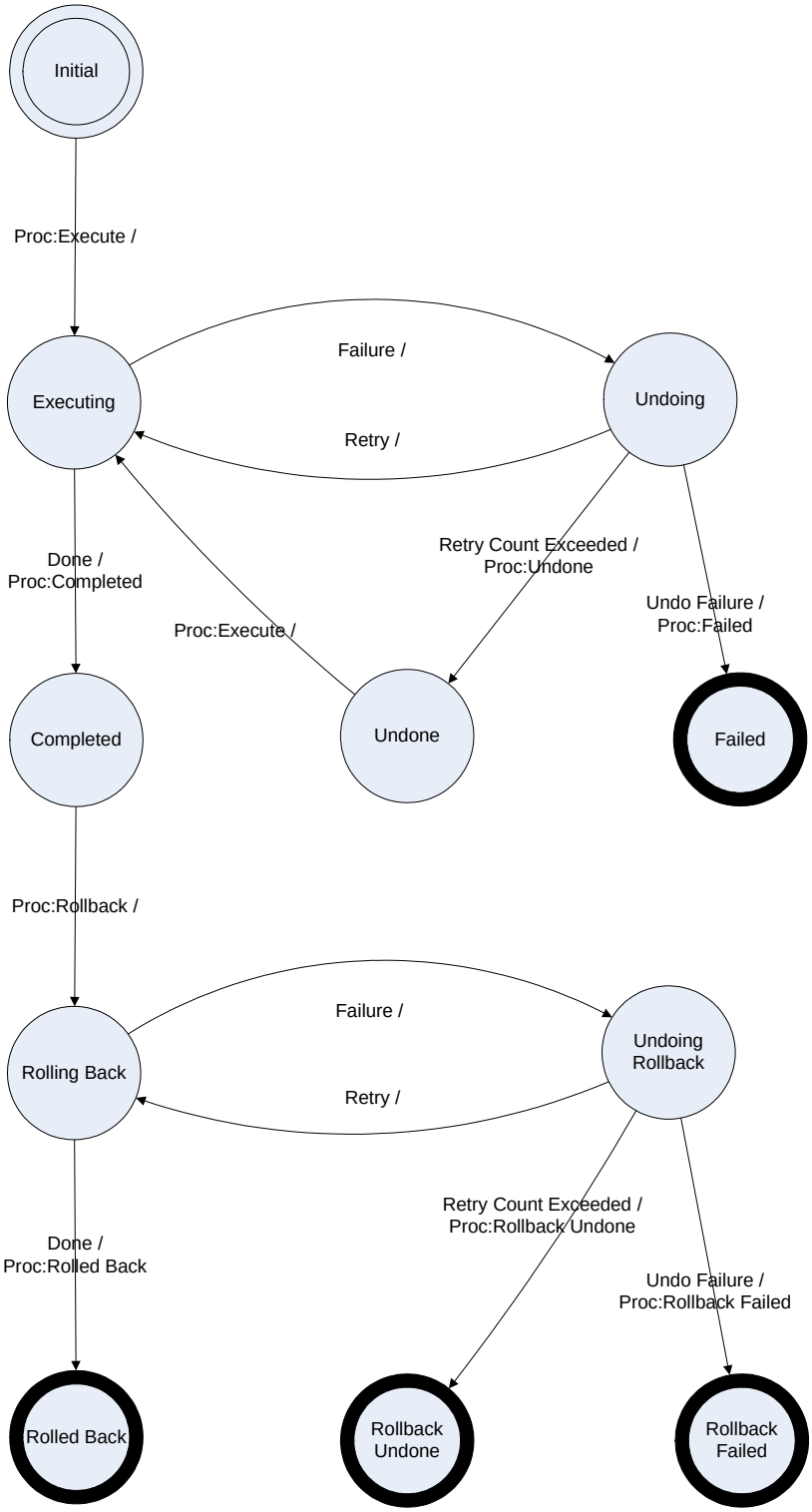
If a failure occurs during execution (Failure), the step FSM moves to the Undoing state.

5.1.3 Completed State

The Completed state is the targeted state for all step FSMs in the campaign; however, it is not a final state, as the campaign may be rolled back by the administrator. Rollback applies only to step FSMs in Completed state.

If a campaign rollback is initiated administratively, this decision is propagated to the step FSM in Completed state as a Proc:Rollback signal, which then moves the step FSM to the Rolling Back state.

FIGURE 5 Upgrade Step State Model Diagram



5.1.4 Undoing State

In the `Undoing` state, the already performed actions of the step are undone according to Section 4.2.2.1.

If the step is undone successfully, and a `Retry` is permitted, the step FSM returns to the `Executing` state and re-executes the actions of the step; otherwise, if the retry count was exceeded (`Retry Count Exceeded`), the step FSM moves to the `Undone` state and signals this transition to the procedure FSM (`Proc:Undone`).

If another failure (an `Undo_Failure`) occurs while undoing the actions of the failed step, that is, while the step FSM is in the `Undoing` state (which means that the failed step cannot be undone), the step FSM moves to the `Failed` state. This transition is communicated to the procedure FSM as a `Proc:Failed` signal.

5.1.5 Failed State

The `Failed` state is a final state for the step FSM representing a double failure situation (that is, the step could not be executed, and the already executed actions could not be reverted).

5.1.6 Undone State

While the step FSM is in the `Undone` state, which is a state similar to `Initial` state, it may receive a `Proc:Execute` signal from the procedure FSM due to an administrative attempt of a forced retry (to allow for cases when the administrator is able to eliminate the root cause of the `Failure` such as some disk space limitation). In this case, the step FSM is moved back to the `Executing` state, and the step is retried.

5.1.7 Rolling Back State

In the `Rolling Back` state, the actions of the step are rolled back according to Section 4.2.2.3.3.

If the rolling back succeeds (`Done`), the step FSM moves to the `Rolled Back` state and reports this transition to the procedure FSM with a `Proc:Rolled Back` signal. The `Rolled Back` state is a final state for the step FSM.

If a failure is detected while rolling back (`Failure`), the step FSM moves to the `Undoing Rollback` state, where the already performed actions are undone as described in Section 4.2.2.3.4. If this undoing succeeds, and a `Retry` is permitted, the step rollback is re-attempted after having transitioned the step FSM back into the `Rolling Back` state.

5.1.8 Undoing Rollback State

If undoing the rollback is successful, but no retry is permitted (`Retry Count Exceeded`), the step FSM moves to the `Rollback Undone` final state and signals `Proc:Rollback Undone` to the procedure FSM. No forced retry is permitted in this state. [1]

If undoing of the actions of the rollback fails (`Undo Failure`), the step FSM terminates in the `Rollback Failed` state while issuing the appropriate `Proc:Rollback Failed` signal to the procedure FSM.

5.1.9 Rolled Back, Rollback Undone, and Rollback Failed States

The `Rolled Back`, `Rollback Undone`, and the `Rollback Failed` states are final states for the step FSM.

5.2 Upgrade Procedure State Model

FIGURE 6 presents the state diagram of the upgrade procedure FSM.

5.2.1 Initial State

The procedure FSM is created in the `Initial` state. The execution of the procedure is initiated by the `Cmpg:Execute` signal received from the upgrade campaign finite state machine, which moves the procedure FSM into the `Executing` state, where it signals a `Step:Execute` to its first step. In addition, the `freeze` internal flag is reset. Since any suspension operation is applied at step boundaries, this flag is set to remember that such an operation was issued while the execution of a step is in progress, and to apply it when the next step boundary is reached.

5.2.2 Executing State

The procedure FSM remains in the Executing state until it receives the outcome of the initiated step from the step FSM. If in this state a `Cmpg:Suspend` is received from the campaign FSM, the `freeze` flag is set, as it is applied only at step boundary.

Depending on the result received from the step FSM, the following transitions become available in the `Executing` state.

1. This is a design choice that was made partially to simplify the FSMs and partially because the system is already in a failure correction mode (i.e., it is rolling back); therefore, faster escalation of the failure is desirable.

1. If the current step reports that it has completed successfully by a `Step:Completed` signal,
   - if there is another step within the procedure that is still in `Initial` state, and the `freeze` flag is not set, a `Step:Execute` is signaled to the next step, which is still in `Initial` state. The procedure FSM remains in the `Executing` state.
   - If there are no more steps in `Initial` state and all of them reached the `Completed` state, and the `freeze` flag is not set, then, after successful verification (see Section 4.1.3), the procedure is also completed, and the procedure FSM moves to the `Completed` state and reports this transition to the campaign FSM with a `Cmpg:Completed` signal. If the procedure verification fails, the procedure FSM moves to the `Failed` state and reports this transition to the campaign FSM with a `Cmpg:Failed` signal.
     Note that setting the `freeze` flag has no impact after the procedure verification has started.
   - If the `freeze` flag is set, the procedure moves into `Suspended` state, reporting its suspension to the campaign with a `Cmpg:Suspended` signal.
2. If the current step reports an error by a `Step:Failed` signal, the procedure FSM reports this failure immediately to the campaign FSM with a `Cmpg:Failed` signal, and the procedure FSM moves to the `Failed` final state.
3. If the current step does not succeed and reports `Step:Undone`, the procedure FSM moves into the `Step Undone` state and reports this transition to the campaign FSM with a `Cmpg:Step Undone` signal.

5.2.3 Suspended and Step Undone States

The procedure FSM remains in the `Suspended` state or in the `Step Undone` state until the campaign FSM signals to which state it shall move next. From both states, two transitions are available, depending on the administrative operation issued at the campaign level.

- If a rollback is selected by the administrator, this is propagated to the procedure FSM by a `Cmpg:Rollback` signal. The procedure FSM shall reset its `freeze` flag and signal `Step:Rollback` to the last step that was completed. It also moves to the `Rolling Back` state.
- If an execute is selected at campaign level, this is propagated by a `Cmpg:Execute` signal to the procedure FSM, which, as a result, transitions into the `Executing` state and resets its `freeze` flag at the same time. In the `Executing` state, if there is a step in `Undone` state, the procedure FSM signals `Step:Execute` to this step; otherwise, it signals `Step:Execute` to the next step which is still in `Initial` state. If there are no more steps, the procedure FSM proceeds with the verification.

5.2.4 Completed State

The Completed state is the targeted state for all procedure FSMs. Ideally, the campaign completes when all of its procedures are in this state.

If the campaign is rolled back, every completed procedure and those that are in the Step Undone or Suspended state shall receive a Cmpg:Rollback signal from the campaign FSM in due time. This moves the procedure FSM into the Rolling Back state while issuing a Step:Rollback to its last completed step.

5.2.5 Rolling Back State

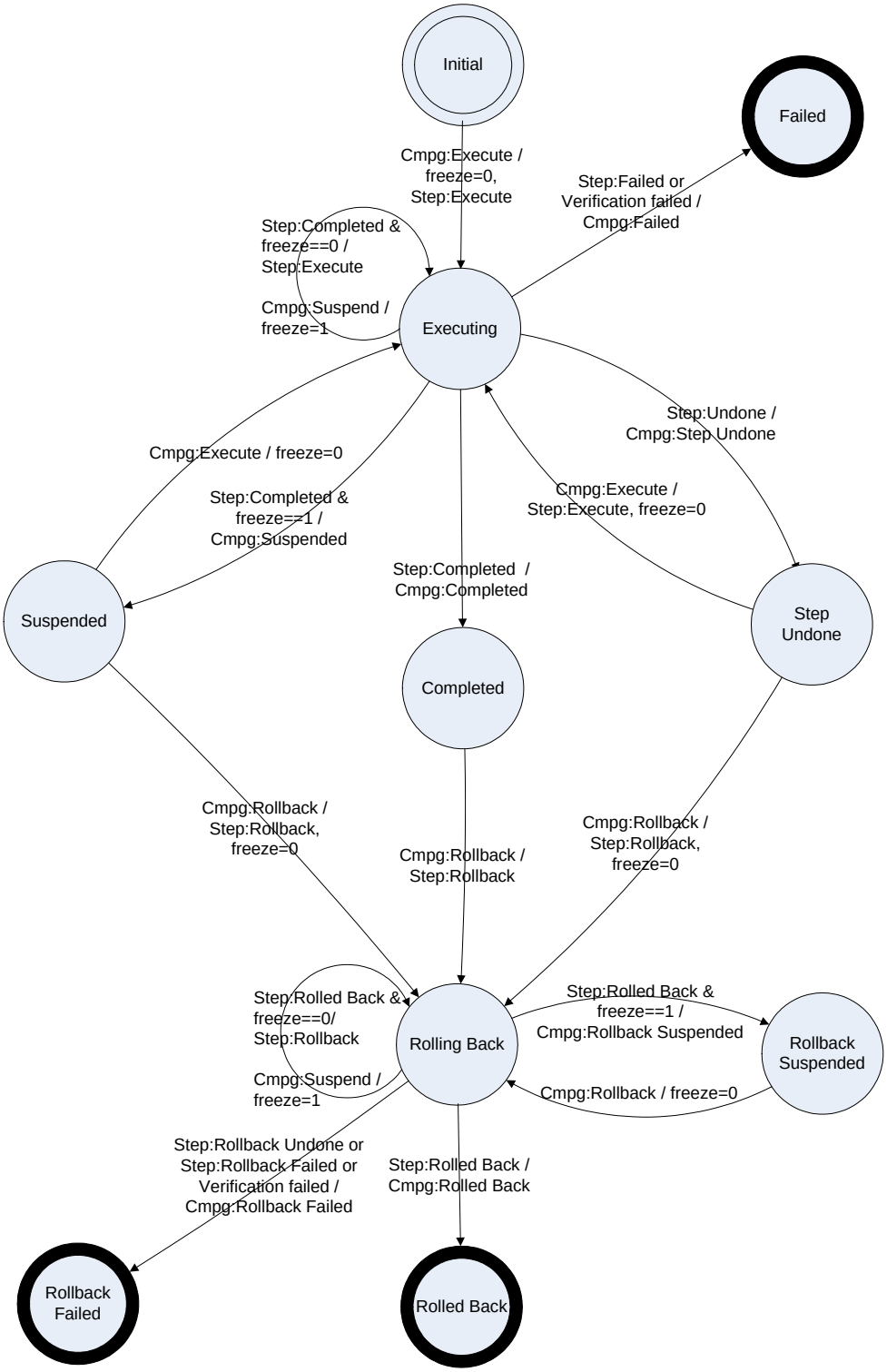
The procedure FSM remains in the Rolling Back state until it receives the result of the currently rolling back step. If in this state a Cmpg:Suspend is received from the campaign FSM, the procedure FSM sets the freeze flag.

If the current step rollback completes successfully, a Step:Rolled Back is received and the freeze flag is not set, then the procedure signals Step:Rollback to its next step which is still in Completed state. If there are no more steps in Completed state, the procedure rollback is completed. Thus, after successful verification (see Section 4.1.3), the procedure FSM terminates in the Rolled Back state and signals this transition to the campaign FSM with a Cmpg:Rolled Back signal. If the verification fails, the procedure FSM moves to the Rollback Failed state and sends a Cmpg:Rollback Failed signal. Both Rolled Back and Rollback Failed are final states.

If the freeze flag was set when the Step:Rolled Back signal was received, the procedure FSM moves to the Rollback Suspended state.

If the step rollback cannot be completed by any reason, a Step:Rollback Undone or Step:Rollback Failed is received by the procedure FSM. This fails the procedure rollback; thus, the procedure FSM terminates in the Rollback Failed state and reports this fact also to the campaign FSM.

FIGURE 6 Upgrade Procedure State Model Diagram



5.2.6 Rollback Suspended State

The procedure FSM remains in the `Rollback Suspended` state until it receives from the campaign FSM a `Cmpg:Rollback` signal permitting it to move back to the `Rolling Back` state to continue the rollback. This transition also resets the `freeze` flag. If there are any steps still in the `Completed` state, a `Step:Rollback` is signaled to the next step. If there is no such step, and depending on the result of the verification, the procedure FSM terminates in either the `Rolled Back` state or in the `Rollback Failed` state and issues the respective signal to the campaign FSM.

5.2.7 Rolled Back, Failed, and Rollback Failed States

The `Rolled back`, `Failed`, and `Rollback Failed` states are final states for the procedure FSM.

5.3 Upgrade Campaign State Model

FIGURE 7 presents the state diagram of the upgrade campaign FSM, which is also created in the `Initial` state.

5.3.1 Initial State

The execution of the campaign is initiated by the EXECUTE[1] administrative operation, which initiates the prerequisite check as described in Section 4.1.1, including the creation of the system backup according to Section 4.2.1.1. If any of the operations fail, the campaign cannot be started and remains in the `Initial` state. The administrative operation returns with the error indicating that the campaign could not be initiated. The execution can be re-attempted at a later time when the cause of the failure has been corrected.

If the prerequisite check including the backup operations succeed, the campaign FSM moves to the `Executing` state while it signals `Proc:Execute` to all procedures of the first execution level (see Section 3.3.3.3).

The campaign FSM remains in the `Executing` state until it receives the outcome from all the initiated procedures or an administrative intervention happens.

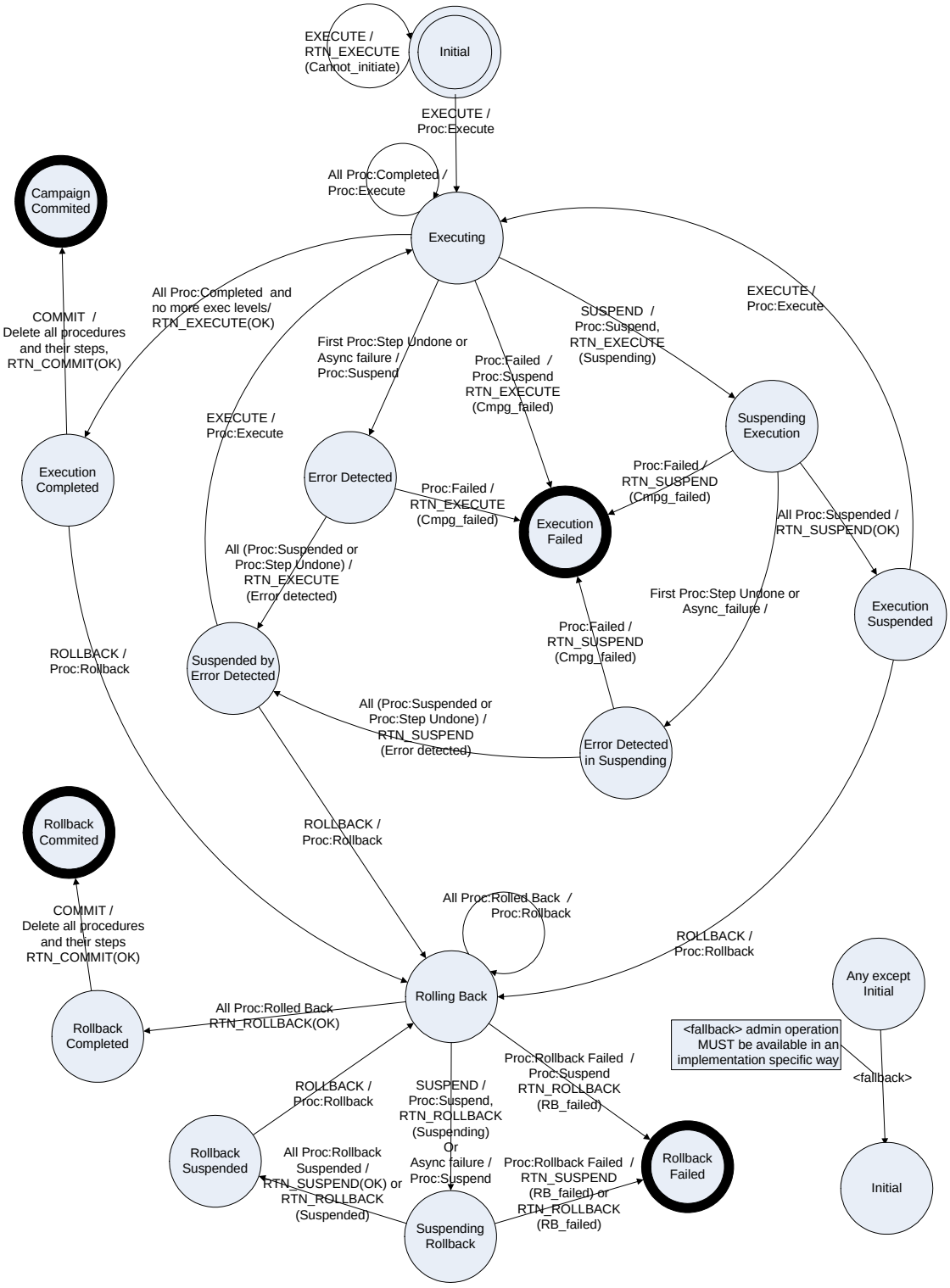
5.3.2 Executing State

Depending on the outcome received from a procedure FSM, the following transitions become available in the `Executing` state:

1. The SA_SMF_ADMIN_ prefix is omitted for all administrative operations.

1. If all the currently executing procedures (that is, all procedures of the current execution level) successfully report `Proc:Completed`,
   - if there is another execution level that was not initiated yet, a `Proc:Execute` is initiated on each procedure FSMs of the next execution level. The campaign FSM remains in the `Executing` state.
   - If there are no more execution levels, and after successful verification, including the consideration of any observation period (see Section 4.1.4), the campaign FSM moves to the `Execution Completed` state. This transition also results in the successful completion of the EXECUTE administrative operation (`RTN_EXECUTE(OK)`). The campaign remains in the `Execution Completed` state until the administrator decides whether the campaign can be committed.
2. If any of the procedures fails and reports a `Proc:Failed` while the campaign still in the `Executing` state,
   - the campaign FSM signals `Proc:Suspend` to all running procedures; additionally, it reports the failure to the administrator by returning the `RTN_EXECUTE(Cmpg_failed)` signal and moves into the `Execution Failed` state.
3. If any of the currently running procedures reports a `Proc:Step Undone`, the campaign FSM moves into the `Error Detected` state and signals `Proc:Suspend` to the procedures still executing to suspend them as well.
4. If the Software Management Framework receives a signal reporting the failure of an AMF entity (`Async_failure`, see also Section 4.2.1.3) while the campaign is in the `Executing` state, the campaign is automatically suspended. As a consequence, the campaign FSM signals `Proc:Suspend` to all currently executing procedures and moves into the `Error Detected` state.
5. If a SUSPEND administrative operation is requested while the campaign FSM is in the `Executing` state, the campaign FSM signals `Proc:Suspend` to all currently executing procedures, interrupts the EXECUTE administrative operation by returning a `RTN_EXECUTE(Suspending)` signal, and moves into the `Suspending Execution` state.

FIGURE 7 Upgrade Campaign State Model Diagram



5.3.3 Execution Completed State

In the `Execution Completed` state, the administrator has the option of committing the campaign by issuing the `COMMIT` administrative operation. This administrative operation triggers the wrap-up operations of the campaign and moves the FSM into the final `Campaign Committed` state, while confirming the `COMMIT` with a `RTN_COMMIT` signal.

Alternatively, the administrator may decide that the campaign cannot be committed and needs to be rolled back. This results in a `Proc:Rollback` signal sent to all procedures of the last execution level, and the campaign FSM moves to the `Rolling Back` state.

While in the `Execution Completed` state, the Software Management Framework still must collect the information on any asynchronous failure of AMF entities as described in Section 4.2.1.3. However these failures trigger no state transition.

5.3.4 Error Detected State

The campaign remains in the `Error Detected` state until one of the two following situations occur.

- A procedure fails and reports a `Proc:Failed`. The campaign FSM reports the failure to the administrator by returning the `RTN_EXECUTE(Cmpg_failed)` signal and moves into the `Execution Failed` state.
- All executing procedures finish and all of them report `Proc:Completed` or `Proc:Step Undone` (i.e. none of them report a `Proc:Failed`). In this case, the campaign FSM moves to the `Suspended by Error Detected` state. This fact is reported to the administrator by returning an `RTN_EXECUTE(Error_detected)`.

It is possible that further AMF entity failures are detected in the `Error Detected` state. These failures trigger no state transition.

5.3.5 Suspending Execution State

If the execution was suspended by the administrator, the campaign FSM remains in the `Suspending Execution` state until one of the following situations occurs.

1. All the running procedures reached the `Suspended` state successfully, and each of them sent a `Proc:Suspended` signal. When all of these signals arrive, the campaign FSM moves to the `Execution Suspended` state and returns an `RTN_SUSPEND(OK)` in response to the `SUSPEND` administrative operation.

2. Any of the procedures reports a failure by the Proc:Failed signal. The campaign returns an RTN_SUSPEND(Cmpg_failed) signal to the administrator, and the FSM moves to the Execution Failed state.
3. Any of the procedures reports Proc:Step Undone, or the failure of an AMF entity is detected. The campaign FSM moves into the Error Detected in Suspending state.

5.3.6 Error Detected in Suspending State

In the Error Detected in Suspending state, one of the following transitions are available.

- if a procedure reports a failure by the Proc:Failed signal, the campaign terminates in the Execution Failed state and returns an RTN_SUSPEND(Cmpg_failed) signal to the administrator;
- if no procedure reports a failure, but all of them reach the Suspended or Step Undone state, the campaign FSM moves to the Suspended by Error Detected state and returns the RTN_SUSPEND(Step_undone) signal to the administrator.

It is also possible that further AMF entity failures are detected in the Error Detected in Suspending state; however, these failures trigger no state transition.

5.3.7 Suspended by Error Detected State

In the Suspended by Error Detected state, the administrator is given the opportunity to fix the problem that prevented the completion of the steps that are in the Step Undone state.

If the reason for reaching this state was the occurrence of one or more asynchronous failures, the administrator needs to decide whether these failures are related to the upgrade campaign or not.

In either case, an attempt to retry the campaign from its current state can be forced by issuing a new EXECUTE administrative operation, which is propagated to all procedures of the execution level as a new Proc:Execute, and the campaign FSM moves back to Executing state.

Alternatively, the administrator may decide that the campaign cannot be completed and issues a ROLLBACK operation. This decision is propagated to all procedure FSMs of the current execution level by Proc:Rollback signals, and it also moves the campaign FSM into Rolling Back state.

Regardless of whether the campaign is retried or rolled back, it is expected that the administrator has completed any necessary repair and enabled the failed entities as appropriate before issuing the administrative operation.

5.3.8 Execution Suspended State

The campaign FSM remains in the `Execution Suspended` state until the next administrative operation determines to which state it should move. If a new EXECUTE administrative operation is issued, it is propagated to all procedures of the execution level as a new `Proc:Execute`, and the campaign FSM moves back to the `Executing` state. Alternatively, a ROLLBACK may be ordered, which is propagated to all procedure FSMs of the current execution level by `Proc:Rollback` signals.

While in the `Execution Suspended` state, the Software Management Framework still must collect the information on any asynchronous failure of AMF entities as described in Section 4.2.1.3. However these failures trigger no state transition.

5.3.9 Rolling Back State

The campaign remains in the `Rolling Back` state until it receives the result from all currently rolling back procedures.

1. If the rolling back of all procedures of the current execution level completes successfully, and a `Proc:Rolled Back` is received from each of them.
   - If there is an execution level lower than the current one, the campaign signals `Proc:Rollback` to all procedures with the next smaller execution level and remains in the `Rolling Back` state.
   - If there are no more execution levels to roll back, and all the verifications, including the consideration of any observation time are successful, the campaign rollback is completed, the campaign FSM moves to the `Rollback Completed` state, which is indicated to the administrator by returning RTN_ROLLBACK(OK) in response to the ROLLBACK operation.
2. While in the `Rolling Back` state, if any of the procedure rollbacks cannot be completed due to a failure that is reported by a `Proc:Rollback Failed` signal, or if the final verification of the rollback fails, the campaign rollback also fails and the FSM moves to the `Rollback Failed` state. This transition is also reported by an RTN_ROLLBACK(`RB_failed`) signal.
3. An administrator may also issue a SUSPEND administrative operation to suspend a rollback. Similar to what happens in the `Executing` state, this administrative operation causes a `Proc:Suspend` signal to be sent to all running procedures, a response of RTN_ROLLBACK(`Suspending`) to be sent the administrator, and the transition of the campaign FSM to the `Suspending Rollback` state.

4. It may happen that during rollback a signal is received that reports the failure of an AMF entity (see Section 4.2.1.3). This signal suspends the rollback; as a result, a `Proc:Suspend` signal is sent to each running procedure, and the campaign FSM moves to the `Suspending Rollback` state.

5.3.10 Rollback Completed State

In `Rollback Completed` state, the administrator still needs to explicitly commit the rollback by issuing a `COMMIT` administrative operation. This triggers the wrap-up operations of the campaign, after which the `COMMIT` is confirmed by an `RTN_COMMIT`, and the campaign FSM moves into the final `Rollback Committed` state.

While in the `Rollback Completed` state, the Software Management Framework still must collect the information on any asynchronous failure of AMF entities as described in Section 4.2.1.3. However these failures trigger no state transition.

5.3.11 Suspending Rollback State

The campaign remains in the `Suspending Rollback` state until

1. all `Proc:Rollback Suspended` signals are received from all running procedures. If so, the campaign moves into the `Rollback Suspended` state. If the rollback was suspended due to an administrative operation, an `RTN_SUSPEND(OK)` is returned; if the rollback was suspended due to an asynchronous error, an `RTN_ROLLBACK(Suspended)` is returned to the administrator.
2. If any of the procedures report a failure by a `Proc:Rollback Failed` signal, the campaign FSM moves to the `Rollback Failed` state and reports this transition with an `RTN_SUSPEND(RB_failed)` or `RTN_ROLLBACK(RB_failed)` signal as appropriate.

While the campaign is in `Suspending Rollback` state, subsequent asynchronous failures may be detected; however, they do not result in a state change.

5.3.12 Rollback Suspended State

To resume the rollback of the campaign in the `Rollback Suspended` state, the administrator must issue a new `ROLLBACK` administrative operation, which is propagated as `Proc:Rollback` signals to procedures at the current execution level, and the campaign moves back to the `Rolling Back` state.

While in the `Rollback Suspended` state, the Software Management Framework still must collect the information on any subsequent asynchronous failure of AMF entities as described in Section 4.2.1.3. However these failures trigger no state transition.

5.3.13 Execution Failed and Rollback Failed States

In the final `Execution Failed` and `Rollback Failed` states, the only possible administrative operation is to initiate a fallback (Section 4.2.2.4).

5.3.14 System Backup, Restart, and Fallback Operations

An upgrade campaign cannot be started without the creation of a system backup. Once an upgrade campaign has been initiated, and the campaign FSM has moved away from the `Initial` state, no new system backup may be created that may overwrite this initial backup until the upgrade campaign is terminated in one of the Committed states: `Campaign Committed` or `Rollback Committed`. The backup operation must be blocked, or the backed up image must be protected in some other ways to ensure that during the campaign it is always possible to return to the initial system state by issuing a system fallback, which requires the backup created when the campaign was initiated.

Until the campaign is committed in the `Campaign Committed` or the `Rollback Committed` state, at any time and in any state other than the `Initial` state, the administrator may order a campaign fallback. This should restore the system to the state saved in the system backup, which was created in the `Initial` state of the campaign. Once the campaign is committed in the `Campaign Committed` or the `Rollback Committed` state, there is no guarantee that a fallback operation restores the system backup taken at the beginning of the campaign. See also Section 4.2.1.1, Section 4.2.2.4, and Section 6.5.2.

It is possible that a cluster restart is initiated outside of an executing upgrade campaign. For example, a cluster restart could be initiated by an administrator in order to recover from a fatal failure. Such a restart should be escalated into a fallback operation except if the campaign FSM is in `Initial`, `Execution Completed`, `Campaign Committed`, `Rollback Completed`, or `Rollback Committed` states. Note that even in these states the restart may result in the loss of some or all the information represented by the upgrade procedure and upgrade step runtime objects.

6 Upgrade Campaign Specification

An upgrade campaign is specified in an XML file that follows the upgrade campaign XML schema provided in [9]. This chapter gives additional information on the intended usage of the XML elements defined in the schema.

The upgrade campaign specification can be perceived as a description of the execution flow of the upgrade campaign, which, besides the procedural elements, also contains the configuration change information embedded in these procedural elements, so they can be applied at the appropriate moment within the execution flow.

An XML file that specifies an upgrade campaign shall contain the `upgradeCampaign` element, which has a name attribute containing the RDN of the campaign object that shall be created from the campaign specification. There are four mandatory elements within the campaign element:

- `campaignInfo`, which provides general information about the upgrade campaign specified in the file,
- `campaignInitialization`, which specifies information necessary for the prerequisite check and also for other preparatory actions that may be necessary for the campaign,
- `upgradeProcedure`, which specifies one or more upgrade procedures of the upgrade campaign, and
- `campaignWrapup`, which specifies conditions for committing the campaign and also actions that may need to be performed to complete the campaign.

In this chapter, the subtree of elements under each of these four elements is referred to as a section.

6.1 Common Elements

The XML elements of this section are used in different parts of the upgrade campaign specification.

6.1.1 Action Element

The action element can be used to specify an administrative operation, an IMM configuration bundle, a CLI command, or a customized callback action.

6.1.1.1 Administrative Operation

The administrative operation element is used to specify any administrative operation that can be executed using the IMM OM-API. These administrative operations are specified by the operation Id and the DN of the targeted IMM object. Optionally, a set of attributes can be specified with their name, type, and value set.

The administrative operations need to be specified in pairs: the `doAdminOperation` element defines the administrative operation executed on the forward path of the upgrade campaign; the `undoAdminOperation` element defines the administrative operations that are executed on the rollback path of the campaign to “undo” the effects of the forward path operation.

6.1.1.2 Configuration Change Bundle

Any Information Model Management Service configuration change bundle (CCB) is described by specifying the individual operations within the CCB. In case of a roll-back, the original content of IMM must to be restored. This specification does not describe how this is achieved by a given implementation. The specification of the operations follow the IMM API specification, and the attributes and elements match the appropriate function signatures.

6.1.1.3 CLI Command

Similarly to the administrative operations, CLI commands need to be specified in pairs for the forward (`doCliCommand`) and the rollback (`undoCliCommand`) paths. These elements specify the command and the args strings as attributes. In addition, the `node` element lists the nodes on which the CLI commands must be executed by the Software Management Framework. The result of a CLI command must be according to Section 4.1.5.

6.1.1.4 Customized Callback Action

The upgrade campaign allows for the specification of a number of callbacks during the campaign. These callbacks are used to report the progress of the upgrade campaign and also to perform application-level actions synchronized with the campaign (see also Section 7.2.3.1 and Section 8.5.1).

Accordingly, each callback specified in the campaign needs to be customized by the following attributes:

- A callback is identified by the mandatory `callbackLabel` attribute, which is a string. This string is matched against the filters registered by the different user processes, and those processes that specified a filter that the label matches

shall receive a callback from the Software Management Framework. The callback label is passed onto the user process in the callback.

- The second optional attribute that is passed in the callback to the registered process is the content of the `stringToPass` string. It is not interpreted by the Software Management Framework, and the string is passed in the callback without any change.
- The optional `timer` attribute specified for the callback action determines whether the Software Management Framework must wait for a response from the called back process and the upper limit of this waiting period. If no timer is specified, the campaign may continue right after the Software Management Framework has issued the callbacks to all interested processes. If a timer is specified the Software Management Framework must wait for the responses for the specified period. If all responses have been received, the campaign may continue according to the results returned in the response. When the timer expires, the missing answers are taken as a success.

When a callback is invoked, the Software Management Framework passes to the process a parameter that identifies whether the callback is performed on the forward or the rollback path of the campaign execution (see Section 8.5.1), and the process uses this parameter to determine the appropriate application-level actions.

6.1.1.4.1 Timing of Customized Callback Actions

The timing of a customized callback within the flow of the upgrade campaign can be specified in different ways.

The obvious way is based on the placement and the order of the customized callback actions within the upgrade campaign specification, which defines also the execution order.

For some cases, such ordering is unfeasible or cannot be determined. For example, it is unknown in advance whether and when a rollback is going to be ordered during the campaign. For such cases, a number of elements are predefined in the XML schema. These elements represent certain conditions or operations within the campaign execution flow that can be associated with customized callbacks. The callbacks are then invoked when the condition is fulfilled. See also Section 7.2.3.1, Section 6.3.3.2.1, Section 6.3.3.2.2, Section 6.3.3.2.3, and Section 6.5.2.

6.2 Campaign Information

The `campaignInfo` element provides general information about the upgrade campaign. This is the only section intended to be human readable, if necessary. Two elements are associated with the `campaignInfo` element, as described in the following subsections.

6.2.1 Campaign Period

The campaign period (`campaignPeriod`) provides an estimate of the time that the whole upgrade campaign is expected to take (see Section 3.4.2). It shall be used by the administrator to schedule the execution of the upgrade campaign in an appropriate maintenance window. During the execution of the campaign, the elapsed time can be compared with this time to decide what administrative action is the most appropriate in a given situation. For example, if a failure is detected when most of the campaign period was already consumed, and, therefore, it is not likely that a rollback can be completed within the available maintenance window, it may be more appropriate to order a fallback.

The Software Management Framework itself is not expected to use the campaign period value.

6.2.2 Configuration Base

It is optional to specify a reference to the deployment configuration (`configurationBase`) based on which this upgrade campaign was created and to which the upgrade campaign should be applied. If the deployment configuration has changed compared to the configuration identified by `configurationBase`, the campaign may not be relevant any more or may even jeopardize availability in some cases; therefore, if such a reference is given in the campaign specification, it must be respected, which means that the Software Management Framework must not proceed with the execution of the campaign if it detects that the configuration has changed compared to the reference configuration. See also prerequisite 5. of Section 4.1.1.

The Information Model Management Service maintains the time stamp of the last configuration change in its content. The `configurationBase` is a time stamp that specifies when the IMM content was exported. This time stamp is used as a basis of the upgrade campaign specification. If the `configurationBase` is set, the Software Management Framework must guarantee that the only configuration change between the time specified by `configurationBase` and the time of the execution of the upgrade campaign was the creation of the upgrade campaign object representing the campaign to be executed.

6.3 Campaign Initialization

6.3.1 Required Software Bundles

The `campaignInitialization` element specifies additional information necessary for the prerequisite check described in Section 4.1.1. In particular, for checking prerequisite 6., the software bundles required for the campaign are listed as `softwareBundle` elements that are to be added to the information model (`addToImm`).

Using this list, the Software Management Framework must check whether the specified software bundle objects exists in the Information Model Management Service. If they are part of the software catalog, it checks in an implementation-specific way whether the associated images indeed are in the software repository. If the bundle objects are not in the catalog, the Software Management Framework needs to verify in an implementation-specific way if the associated bundles are available in the software repository, and, if this is the case, the Software Management Framework adds the corresponding objects to the software catalog. If any of the checks fails, the upgrade campaign may not proceed.

Notice that the information provided to identify the required bundles is similar to the information that is expected from the software vendors in the entity types XML file (see Section 7.1.1). However, the `installation` and `removal` elements shall contain all the information adjusted to the current deployment configuration of the target system and not necessarily the information that was provided by the vendor (Section 7.1.1.2). In addition, a default timeout (`defaultCliTimeout`) that is appropriate for the system is defined for these CLI operations.

6.3.2 New AMF Entity Types

As part of the campaign initialization, the new software entity types are added to the information model. Currently, this includes only AMF entity types (`amfEntityTypes`) within the `addToImm` element.

The XML schema for new AMF entity types specified in the upgrade campaign XML is NOT the same as the one defined for the entity types file, which specifies entity prototypes!

As discussed in Section 7.2.2, the attributes that need to be specified for the deployment configuration are more specific than those expected from the software vendor. While the vendor only needs to specify the boundary conditions for deployment, the AMF entity types contained in the information model specify configuration values for common attribute and default values for all of their entities.

The `amfEntityTypes` element specifies the XML schema for the AMF entity types to be added to the system's information model in accordance with the object class definitions of the Availability Management Framework [2]. Whenever possible, the attribute name in the schema was chosen to be the same as the attribute name of the UML model for which the attribute in the schema provides the values.

6.3.3 Initialization Actions

The campaign initialization section also includes any actions that may be required to prepare for the execution of the campaign. It allows for a more generic action element and for three predefined conditions for customized callbacks.

6.3.3.1 Generic Initialization Action

The `campInitAction` element can be used to specify:

- any administrative operation,
- any Information Model Management Service configuration change bundle (CCB),
- any CLI command, or
- any customized callback,

as described in Section 6.1.1.

6.3.3.2 Predefined Conditions for Customized Callback Actions

Three conditions are predefined in the campaign initialization section. These conditions are associated with customized callback actions. All of them are optional. The customized callback specification must follow Section 6.1.1.4.

6.3.3.2.1 Callback at Campaign Initialization

The campaign initialization callback actions (`callbackAtInit`) are used to specify callbacks that signal to registered processes of upgrade-aware entities that an upgrade campaign is about to start. If such an action is specified in the upgrade campaign specification, then when an upgrade campaign is initiated, the Software Management Framework shall call back all processes that registered their interest in the campaign initialization. The campaign shall not start until all of them indicated their consent to the campaign. The time limit specified is used by the Software Management Framework to delay the campaign: it will wait for an answer at most for this period. If no response is received, it is taken as a consent. Negative answers may be taken by an implementation as a reason to terminate the campaign (Section 4.1.1, prerequisite 10.). If no time limit is specified, the campaign shall commence without waiting for the responses.

A callback is also invoked on a process that registers one of the specified callback labels when the campaign is already in progress. However, in these circumstances, the time limit and any response from the process are ignored, as the campaign is already in progress.

6.3.3.2.2 Callback at Campaign Backup Creation

The campaign callback actions (`callbackAtBackup`) allow the invocation of a callback to solicit applications to create their own application-level backup, synchronized with the upgrade campaign. It is also specified whether the Software Management Framework shall wait for application backups to complete and the maximum waiting period. Failing to receive a response within the waiting period does not prevent the campaign to proceed. Only a negative response fails the associated prerequisite check (Section 4.1.1, prerequisite 11.).

6.3.3.2.3 Callback at Campaign Rollback

The campaign rollback callback actions (`callbackAtRollback`) allow one to specify callbacks that are invoked when the campaign is rolled back due to an administrative decision (see Section 4.2.2.3, Section 5.3.8, Section 5.3.7, and Section 9.3.4). If these actions are specified, and the administrator orders the campaign to rollback, the Software Management Framework calls back registered processes to inform them about the initiation of the rollback. It waits for responses for at most the time specified for the callback. If a negative response is received, the Software Management Framework must interpret it as a failure in the execution of the rollback as described in Section 5.3.9 and move the campaign state machine to the `Rollback Failed` state.

6.4 Campaign Body

The campaign body consists of the specification of a set of upgrade procedures (`upgradeProcedure`). Each procedure has a name and an execution level attribute. Within each procedure, two mandatory elements are expected, the outage information (`outageInfo`) and the upgrade method (`upgradeMenthod`).

In addition, one may optionally specify procedure initialization (`procInitAction`) and procedure wrap-up (`procWrapupAction`) actions according to Section 6.1.1. In particular, procedure wrap-up actions can be used for the verification of the procedure by executing scripts or invoking customized callbacks. These optional elements are not discussed further in this document.

6.4.1 Outage Information

The outage information specifies the acceptable service outage (`acceptableServiceOutage`) in terms of service instances that may become unassigned during the execution of the given upgrade procedure. This information is used at the check of prerequisite 9. of Section 4.1.1. Based on the current assignment state of service instances in the system and the deactivation units of the procedure, the Software Management Framework calculates whether the acceptable service outage can be met for each procedure within the campaign.

During the campaign, the outage information may be used when the Software Management Framework reacts to failures reported by the Availability Management Framework. The campaign should not be suspended due to the Availability Management Framework failing to assign an SI during a procedure for which this SI is allowed to become unassigned.

The acceptable service outage is specified as one of the following options:

⇒ `all`

Allows for any SI and all SIs to become unassigned during the upgrade procedure; this option is typically used for urgent and/or mandatory upgrades where the necessity of the upgrade overrides the availability requirements.

⇒ `none`

Does not permit any service instance to become unassigned. Note that it still allows for SIs to become partially assigned. This option is appropriate for upgrades that are not urgent and that should take place only if it is expected that no SI during any procedure will become unassigned. If it is expected that any SI may become unassigned during the campaign, the campaign must not start.

⇒ a combination of

- `max`

  The maximum number of SIs that may become unassigned from all SIs, or from those not mentioned in the `mustKeepAssignedSI` list, or from those on the `mayGoUnassignedSI` list, if specified. In the last case, `max` must be less than the size of the list. The `max` option is typically used when the SIs are equally ranked. If the Software Management Framework determines that a higher number of SIs will become unassigned than indicated by this value, the campaign must not start.

- and a choice of
  - `mustKeepAssignedSI`

    The list of service instances that are not allowed to go unassigned during the procedure. In addition, the number of SIs that may go unassigned may be limited by `max`. If the Software Management Framework determines that any SI from the list will become unassigned, the campaign must not start.

  - `mayGoUnassignedSI`

    The list of service instances that are allowed to go unassigned during the procedure. `mayGoUnassignedSI` may be used together with `max`. If the Software Management Framework determines that any SIs other than those on the list will become unassigned, the campaign must not start.

These calculations need to be carried out by the Software Management Framework before starting the execution of the campaign, that is, before it initiates the execution of any of the procedures within the campaign (see also Section 3.3.4) as part of the check of prerequisite 9. of Section 4.1.1.

The second element, the procedure period (`procedurePeriod`), is primarily informative and is used to evaluate the expected outage time for those SIs that may become unassigned and also to be able to judge the progress of the campaign during execution (see also Section 3.4.1).

6.4.2 Upgrade Method Specification

The current XML schema allows for specifying upgrade procedures using either the rolling upgrade (Section 3.3.1.1.1) or the single-step upgrade (Section 3.3.3.2.2) methods. Each of these upgrade methods requires the specification of the upgrade scope (Section 3.3.3.1) and the upgrade step (Section 3.3.2) options.

The `saSmfProcDisableSimultanExec` attribute of the `rollingUpgrade` element indicates whether procedure optimization - that is, simultaneous execution of upgrade steps - is permitted for the procedure being specified.

6.4.2.1 Specification of Rolling Upgrades

Currently, the upgrade scope for rolling upgrades is specified by templates. A template is based on similarities; therefore, it can be used to select entities of a symmetric scope (Section 3.3.3.1), which also means symmetric activation units (Section 3.3.2.2).

The template-based schema can be used to specify the upgrade of existing entities. Their logical identity remains the same within the information model, but they will become instances of a different AMF entity type, and, accordingly, some of their configuration attributes are also modified by the upgrade.

The following items must be specified for existing entities:

- the target node(s) and the software bundles to be installed and uninstalled on them,
- the symmetric activation units (that is, the deactivation unit and the activation unit contain the same set of entities), and
- the configuration changes for the entities targeted by the upgrade step.

6.4.2.1.1 Target Node

The target nodes for the installation and uninstallation operations are defined by the AMF cluster or by an AMF node group to which the nodes (as AMF nodes) belong. The appropriate name needs to be specified in the attribute of the `targetNodeTemplate` element.

On each of the nodes selected this way, the software bundles in the `swRemove` elements must be uninstalled, and those in the `swAdd` elements must be installed at the path indicated by the `pathPrefix` attribute at the path indicated by the pathPrefix attribute.

By default, the AMF node is also the symmetric activation unit for the rolling upgrade specified through templates. In this case, the administrative operations defined for the upgrade step (Section 3.3.2.3) are also applied to the AMF node. The number of nodes determines the number of upgrade steps that need to be performed within the procedure, as each step shall upgrade one node at a time.

If the activation unit is not the entire AMF node, the `activationUnitTemplate` element (see next section) can be used.

6.4.2.1.2 Activation Unit Template

The actions requiring administrative operations within an upgrade step are applied to the entities listed in the symmetric activation unit belonging to the upgrade step. Thus, the symmetric activation unit template should only identify entities on which the appropriate administrative operations are valid.

The `activationUnitTemplate` element specifies the selection criteria for entities composing the symmetric activation units within AMF nodes. They can include enti-

ties selected based on their type (`type`), their parent (`parent`) or the combination of these.

1. Use of the Parent Element Only

In the `parent` element, only a service group can be specified. If no type is specified, the activation units will include the service units of the service group. In this case, the upgrade procedure must upgrade one service unit at a time within the service group. This means that a separate upgrade step is performed for each service unit if there are multiple service units on the same node. The installation and uninstallation operations of these steps will target the same node, while the administrative operations will target each service unit on the node separately, as each of them represents a separate symmetric activation unit. Note that service units of different service groups can be collected together into the same symmetric activation unit, which then will include one service unit of each service group.

It is also possible that there is no service unit of the specified service group hosted on a node identified as a target node. In this case, still the installation and uninstallation operations must be performed for these nodes. The order of nodes for these operations is undefined and irrelevant as, by definition, the activation unit must encompass all the entities with which these operations may interfere and there are no activation units on these nodes.

2. Use of the Type Element Only

In the `type` element, a service unit type or a component type can be specified. If no parent is specified, the symmetric activation units will be the service units or the component of the specified type. Appropriately, if there are multiple entities of the same type collocated on the node, the upgrade procedure must upgrade one service unit or component of the specified type at a time. This means that there should be a separate upgrade step performed for each such symmetric activation unit. Just as in case of the parent element, the installation and uninstallation operations of these steps will target the same node, while the administrative operation will target different symmetric activation units.

Again, components of different component types or service units of different service unit types may be collected together into the same symmetric activation unit. However, component types and service unit type must not be used together because a component type must be used together with the upgrade step's restart option, as the only administrative operation available on AMF components is the restart operation. This means that the upgrade step on these symmetric activation units is executed according to the reduced set of actions as specified in Section 3.3.2.3.

It may also happen that there is no entity of any of the specified types hosted on a node identified as a target node. The installation and uninstallation operations must be performed for these nodes as well, but the order of nodes for these operations is undefined and irrelevant.

3. Use of the Type and the Parent Elements Together

The `parent` element which defines a service group may be used together with the `type` element specifying a component type. In this case, the activation units are composed of the components of the specified type within the service units identified by the service group. This option must be used together with the upgrade step's restart option, as the only administrative operation available on AMF components is the restart operation, and the upgrade step is executed with the reduced set of actions (Section 3.3.2.3).

It may also happen that there is no component satisfying the criteria hosted on a node identified as a target node. The installation and uninstallation operations are performed for these nodes as well, but the order of nodes for these operations is undefined and irrelevant.

4. Example

The following example demonstrates the use of these schema elements:

There are four nodes N1, N2, N3, and N4 belonging to a node group NG. Two service groups are distributed on this same node group. SG1 has three service units {SU1, SU2, SU3} of type SUT1 that are hosted on nodes N1, N2, and N3 respectively. Service Group SG2 has five service units {SU4, SU5, SU6, SU7, SU8} of type SUT2. SU4 is hosted on N1, SU5 on N2, SU6 on N3 and SU7 is collocated with SU8 on node N4. SUT1 and SUT2 are built from the same component type CT1, SUT1 having two instances of it in each service unit with RDNs c1 and c2, SUT2 allowing one instance with an RDN c.

Different symmetric activation units can be specified depending on the use of the different schema elements (see Table 3):

Table 3 Valid Symmetric Activation Unit Specifications Using Templates

Target node template	Parent element	Type element	Number of upgrade steps	Entities in the symmetric activation units
NG	-	-	4	{N1}, {N2}, {N3}, {N4}
NG	SG1	-	4	{SU1}, {SU2}, {SU3}, {}
NG	SG2	-	5	{SU4}, {SU5}, {SU6}, {SU7}, {SU8}
NG	SG1 SG2	- -	5	{SU1, SU4}, {SU2, SU5}, {SU3, SU6}, {SU7}, {SU8}
NG	-	CT1	11	{SU1/c1}, {SU2/c1}, {SU3/c1}, {SU1/c2}, {SU2/c2}, {SU3/c2}, {SU4/c}, {SU5/c}, {SU6/c}, {SU7/c}, {SU8/c}
NG	SG2	CT1	5	{SU4/c}, {SU5/c}, {SU6/c}, {SU7/c}, {SU8/c}

Note that in the second row there is potentially an additional upgrade step that installs and uninstalls the necessary software bundles on the fourth target node, which hosts no symmetric activation unit.

Since the last two rows identify activation units that are composed of components only, the upgrade step must specify the restart option.

6.4.2.1.3 Target Entities

The target entity template element specifies the configuration updates that need to be in place for each upgrade step to succeed. Thus, the configuration updates may target any entity that is represented in the Information Model Management Service. Appropriately, it should be possible to select any such entity and specify the required modifications.

Within each symmetric activation unit, the entities targeted by the upgrade may only be a subset of the activation unit. These are the entities the configuration of which needs to be changed. Therefore, the target entities may be selected in addition to the specification of the activation unit, but if the symmetric activation units coincide with the entities that are being upgraded in each step, this selection criteria can be omitted. However, the template for the IMM modification operation still must be specified.

As in case of the symmetric activation unit, there are three ways to specify the targeted entities: By the parent entity, by the entity type, or by the combination of these. These schema elements work in the same way as shown for the activation units.

For example, to upgrade the configuration of the service units of a particular service group, their parent (the service group) can be specified. Alternatively, if configuration of components of a particular type need to be modified on each node or in each service unit (depending on the symmetric activation unit), the component type can be given. Finally, if a service unit type used in a service group is composed of multiple component types, from which only one component type is being upgraded, then both the parent (the service group) and the type (the component type) can be specified as target for the IMM modifications described in the template.

For each such target entity set specified, the IMM modification operations are given in the `modifyOperation` elements.

There could be multiple target entity templates within a scope. The Software Management Framework needs to calculate for each symmetric activation unit the entities identified by the different target entity templates and apply all the modifications within the step.

6.4.2.1.4 Update Template

The update template specifies the modification of the attributes of the upgraded entities. This is given in the `modifyOperation` element. For each target entity template a set of modifications can be specified.

The IMM API for modification requires the object name; therefore, the `modifyOperation` element allows for an object DN attribute. However, since the target entity template identifies the entity to be modified, it does not need to be given. For such an entity, only the attribute modifications need to be given by specifying the operation, the attribute, and its value(s). These modifications are applied to each target entity identified by the template on each upgrade step.

For a target entity that is a compound entity, modifications of the member entities can be specified among the modifications for the target entity itself. In this case, however, the name of the modified member entity needs to be given, which must be an RDN relative to the compound entity. Note that this RDN must be applicable to any target entity selected by the given target entity template.

For example, when the target entities are selected as the service units of a service group, it is possible that not only the version attributes of the service units, but also the version attributes of the components of these service units need to be modified

during the upgrade. For the service units, the `modifyOperation` shall not specify the name attribute, as they are identified by the service group, and on each step, only one service unit is upgraded. However, to identify the component whose version attribute needs to be modified, the `modifyOperation` element may specify the RDN relative to the service unit's name. Alternatively, if there is only one component of the given type in each service unit, it may be selected by using the parent and the type elements together in the selection criteria.

6.4.2.1.5 Upgrade Step of a Rolling Upgrade

The upgrade step of the standard rolling upgrade method includes two options:

- the retry option, which specifies the number of times the upgrade step may be retried in case of failure (Section 4.2.2.2);
- the restart option, which indicates which set of actions of the upgrade step is applied (Section 3.3.2.3). If the option is set, the simplified set of actions is used in each upgrade step, which simply restarts the symmetric activation unit after the modifications are applied to the information model. This simplified upgrade step does not go through the full locking-termination-instantiation-unlocking cycle. This option is applicable if the target entities require no offline installation and uninstallation operations and they are restartable.

Additionally, a number of customized callbacks may be specified for the upgrade step, which is done by specifying the time a callback needs to be issued (`customizationTime`) and the callback itself (`callback`). Each required callback is specified as described in Section 6.1.1.4. The timing is determined by the associated predefined condition elements `onStep` and `atAction`, which are described in the next subsection.

6.4.2.1.6 Timing of Callback Actions Within the Procedure

The `onStep` element specifies at which step the callback is issued within the procedure. The `atAction` specifies at which action of the selected step the callback is invoked.

The `onStep` element can be one of the following choices:

- on each step (`onEachStep`), in which case the callback label is used multiple times within the upgrade procedure, so the registered process needs to distinguish the steps if necessary by the DN of the upgrade step object, which is also provided as a parameter in the callback,
- on the first step (`onFirstStep`), in which case the callback is issued only once on the first iteration of the upgrade step,

- on the last step (`onLastStep`), in which case the callback is issued only once on the last iteration of the upgrade step,
- half way during the procedure (`halfWay`), in which case the callback is issued only once on the iteration that is calculated as *int(N/2)+1* step of the upgrade procedure.

Within a given step, the callback may be issued (`atAction` element) at one of the following situations:

- before the deactivation unit is locked (`beforeLock`),
- before the deactivation unit is terminated (`beforeTermination`),
- after the information model has been modified (`afterImmModification`),
- after the restart or instantiation of the activation unit (`afterInstantiation`), or
- after the activation unit was unlocked (`afterUnlock`).

If the restart option of the upgrade step is set, only the `afterImmModification` or the `afterInstantiation` options are applicable.

6.4.2.2 Specification of Single-Step Upgrades

The single-step upgrade method specification is intended to be used when new entities are added to the system and/or when old entities are removed from the system's deployment configuration. The single-step upgrade method can also be used to upgrade existing entities, in particular, when locking the upgrade scope does not create a service outage.

A procedure of a single-step upgrade is specified by defining its upgrade scope (`upgradeScope`) and upgrade step (`upgradeStep`) options, regardless of whether it modifies existing entities or adds new and/or removes old entities.

6.4.2.2.1 Deactivation Unit Specification

The entities to be removed are specified in the deactivation unit portion of the upgrade scope description of the `forAddRemove` element.

The deactivation unit is specified by the `actedOn` element either as a list or as a template of entities that compose the deactivation unit. The lock and termination operations are applied to all of the entities listed and/or matching the template as described in Section 3.3.2.3.

Within the deactivation unit, the entities that need to be removed are again either listed explicitly in the `byName` element or described by a template in the `byTemplate` element. These entities are removed from the information model.

For the uninstallation of the relevant software, the software bundles must be specified in the `swRemove` elements. Additionally, for each bundle, the list of nodes on which these operations are performed may be given by specifying at least one of the AMF node ([2]) or the CLM node ([5]). If no node is specified, the software bundles need to be removed from all the nodes that host any entity included in the deactivation unit (by the `actedOn` element).

6.4.2.2.2 Activation Unit Specification

The entities to be added to the system configuration are specified in the activation unit portion of the upgrade scope description of the `forAddRemove` element.

The activation unit is specified by the `actedOn` element either as a list or as a template of entities that compose the activation unit. The instantiation and unlock operations are applied to all of the entities listed and/or matching the template of the `actedOn` element (see also Section 3.3.2.3).

Within the activation unit, the entities that are being added are completely specified in the `added` element. This specification provides the name of the parent object in IMM, the type of the IMM object that represents the entity, and the values for each of the necessary attributes. Based on the specification, the Software Management Framework shall create these objects using the IMM OM-API. They should be added to the model in the locked-instantiation administrative state.

Again, the software bundles (`swAdd`) must be specified for the software installation. For each bundle, the list of nodes on which these operations are preformed may be given by specifying at least one of the AMF node ([2]) or the CLM node ([5]). If no node is specified, the software bundles need to be installed on all the nodes that host any entity included in the activation unit by the `actedOn` element.

6.4.2.2.3 Symmetric Activation Unit Specification

The entities to be modified in the system configuration are specified in the activation unit portion of the upgrade scope description of the `forModify` element.

The symmetric activation unit itself is specified by the `actedOn` element either as a list or as a template of entities that compose the activation unit. The lock, termination, instantiation, and unlock or the restart operations are applied to all of the entities

listed and/or matching the template as described for the upgrade step (see Section 3.3.2.3).

The software bundles to be removed (`swRemove`) and to be installed (`swAdd`) must be specified. They will be removed from and/or installed on all the nodes that host any entity included by the `actedOn` element.

Within the symmetric activation unit, the entities targeted by the upgrade may only be a subset of the activation unit. Therefore, the target entities may be selected in addition to the specification of the activation unit. If the symmetric activation unit coincides with the entities that are being modified by the single-step upgrade, the parent and type elements of the `targetEntityTemplate` can be omitted. However, the template for the IMM modification operation still must be specified.

The target entity template element is the same used in the rolling upgrade method (see Section 6.4.2.1.3). As discussed earlier, there are three ways to specify the targeted entities: by the parent entity, by the entity type, or by a combination of these. For each such target entity set specified, the IMM modification operations are given in the `modifyOperation` elements, as described in Section 6.4.2.1.4.

There could be multiple target entity templates for the single step. For the symmetric activation unit, the Software Management Framework needs to determine the entities identified by the different target entity templates and apply all the modifications within the step.

6.4.2.2.4 Upgrade Step of a Single-Step Upgrade

The upgrade step of the standard single-step upgrade method includes a single option, the retry option, which specifies the number of times the upgrade step may be retried in case of failure (Section 4.2.2.2).

It may also specify customized callbacks as described in Section 6.1.1.4. As in this case only one step is performed, the timing needs to be given within the step by specifying the `atAction` element (see Section 6.4.2.1.6) only.

6.5 Campaign Wrap-Up

The campaign wrap-up section specifies any additional actions and timing that may be necessary for the campaign to be considered completed and committed.

The actions may be part of the campaign verification, actions reversing some of the effects of actions performed at campaign initialization or otherwise necessary to leave

the system in a consistent state, or actions related to some cleanup after the completed campaign.

The timing allows for a staged execution of the campaign wrap-up while still different levels of error recovery are possible. It requires the specification of two waiting timers, `waitToComplete` and `waitToAllowNewCampaign`, which are explained in the following subsections.

6.5.1 Completion of the Upgrade Campaign

Even if all the procedures of the campaign have been executed, in many cases, the campaign cannot be considered completed and cannot be committed right away: some actions may be required to verify the results or move the system into a consistent state.

Actions that are required for the decision whether the campaign may be committed must be specified in the `campCompleteAction` element and must be performed after all procedures were successfully executed.

In addition, a waiting period may be specified during which the system is under observation, and the campaign state is still in the `Executing` state; hence, the detected failures are still correlated with the campaign and may result in moving the campaign into a failure state. Accordingly, all the protective measurement taken for the campaign are kept intact for this time. This observation period is specified by the `waitToComplete` element.

The `waitToComplete` timer is started after the `campCompleteAction` was successfully executed. The commit administrative operation may only be issued after the `waitToComplete` timer expired. It is valid to set this timer to zero.

When `waitToComplete` time elapses, the following actions take place:

- the campaign is considered completed, its state changes to Execution Completed, as described in Section 5.3.3, or to Rollback Completed, as described in Section 5.3.10;
- the administrator can initiate the commit operation (refer to Section 4.1.4 and Section 9.3.2).

Once the commit operation has been initiated, resources that would allow campaign rollback (Section 4.2.2.3) are released. (This includes the resetting of the maintenance status of AMF service units.) However, a fallback (Section 4.2.2.4) operation still remains possible.

6.5.2 Committing the Upgrade Campaign

An optional customized callback action (`callbackAtCommit`) has been predefined to specify the need to inform registered processes that the commit administrative operation has been issued. It is specified according to Section 6.1.1.4.

Other actions that are not decisive with respect to committing the campaign may also be performed after the campaign has been committed. These actions are specified in the `campWrapupAction` element, which follows Section 6.1.1. The outcome of these actions have no impact on the campaign's success; however, they may impact the system's normal operation.

Additionally, the second timer (`waitToAllowNewCampaign`) may specify a further time period that shall elapse before all the remaining resources associated with the campaign are freed up. During this time period, the Software Management Framework cannot correlate errors with the campaign in any more; however, the fallback operation is still available, should an error associated with the campaign occur. The fallback operation must be triggered by an administrator. This time period also blocks the initiation of any new upgrade campaign that, for example, could overwrite the backup created for this campaign and so preclude the fallback.

Once the `waitToAllowNewCampaign` timer expires, a new system backup operation becomes available, enabling this way the initiation of a new campaign. When this time elapses, the information model is cleaned up and the IMM objects that are not necessary in the system configuration are removed. In particular, the software bundle objects and the entity type objects that represent software that is not in use any more are deleted from the Information Model Management Service. The objects to be removed are listed by their name in the `removeFromImm` element.

If the `waitToAllowNewCampaign` timer is zero, the associated actions (`campWrapupAction` and `removeFromImm`) are completed as soon as the administrator issues the commit operation.

7 Entity Types File

This chapter presents the XML schema for software bundles and for AMF entity prototypes to be used by software vendors to describe their SA Forum-related product in entity types files [8].

The AMF entity prototypes in the entity types file describe the software implementation from the perspective of its integration with the Availability Management Framework. That is, the prototypes provide all the information necessary to configure the derived AMF entity types (in IMM) and their entities using this software as required by Availability Management Framework.

The assumption is that if a software implementation has no constraints with respect to an AMF configuration attribute or feature, that is, any configuration (value) is acceptable, then there is no need to provide any information for that feature or attribute in the entity types file. Hence, prototypes are only partially specified types. Information is required when particular values or value ranges are expected in the configuration, or if there is any limitation on the composition of the entities of the same or different entity types. Accordingly, the XML schema defined here for the AMF entity prototypes covers the configuration attributes and features of the different AMF entity types; however, many of the XML elements and attributes are left as optional, meaning that if the element or attribute is not specified by the vendor, then any configuration value or any arrangement of entities are permitted by the software implementation.

The subsequent sections introduce the use of the XML schema for the AMF entity prototypes from this perspective. This introduction remains at a high level. More details can be found in the entity types file XML schema.

7.1 Software Bundle

Software bundles must have a unique distinguished name (DN), as described in Section 3.2.4.

7.1.1 XML Schema for Software Bundles

7.1.1.1 Bundle Identification

Within the entity types file (see Section 3.2.5), each software bundle's unique name is used when entity prototypes refer to the bundle.

7.1.1.2 Bundle Handling Operations

Section 3.2.3 presented the bundle handling operations and their use. For each software bundle, these operations are specified as part of the software bundle descriptor. The attributes are defined in this section.

The operations are grouped as installation and removal operations. For each group, the online and the offline portion is specified separately. Each of these portions contains a CLI command, which is specified by two strings, one for the command name along with the relative path, and another one for the command line arguments.

In addition, the **scope of disruption** must be provided for the offline operations by choosing one from the following options:

- Hardware element—the operation may affect any entity within the hardware element, such as the physical node (for instance, hardware reboot).
- Execution environment—the operation may affect any entity within the execution environment of the logical node (for instance, operating system restart).
- CLM node—the operation may affect any entity within the CLM node (for instance, CLM node lock, see [5]).
- AMF node—the operation may affect any entity within the AMF node (for instance, AMF node restart, see [2]).
- Service unit—the operation may affect any entity within the service unit (for instance, service unit restart, see [2]).

The assumption is that if an offline operation is necessary, it will at least affect a service unit.

7.1.1.3 Schema Summary

The following table summarizes the XML schema elements for the software bundle, namely their presence and their attributes. For each element, the parent element is given and a cross-reference is specified if the element is used in conjunction with or as an alternative to other elements. The presence is specified as M—mandatory,

O—optional, or A—alternative. In the latter case, the Cross-Reference column refers to the items among which the choice shall be made.

Table 4 XML Schema Elements of the Software Bundle Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`swBundle`	M	name	M		-
2	`removal`	M			1	-
3	`offline`	O			2, 11	-
4	`command`	M			3, 10	-
5	`args`	M			3, 10	-
6	`serviceUnit`	A			3	6-10
7	`amfNode`	A			3	6-10
8	`clmNode`	A			3	6-10
9	`executionEnvironment`	A			3	6-10
10	`hardwareElement`	A			3	6-10
11	`online`	O			2,11	-
12	`installation`	M			1	-

7.2 AMF Entity Types and their Prototypes

7.2.1 Naming and Versioning

The basic information model of the software catalog is refined for software entity types of the Availability Management Framework, as shown in the following table.

Table 5 AMF Entity Types Specification

Base Entity Types	Versioned Entity Types	Entities
component service base type	component service type	component service instance
service base type	service type	service instance
component base type	component type	component

Table 5 AMF Entity Types Specification (Continued)

Base Entity Types	Versioned Entity Types	Entities
service unit base type	service unit type	service unit
service group base type	service group type	service group
application base type	application type	application

The main role of the base AMF entity types is to group versioned AMF entity types that belong together by some criteria. Some of these criteria may be reflected by some SA Forum-defined attributes; however, in many cases, these criteria are implementation-specifics that are not represented in any of such attributes.

The grouping of the AMF entity types is reflected primarily by their name.

The name of an AMF base entity type is a DN of the following format:

```
safXXXType=<name>
```

Where XXX stands for `Comp`, `Su`, `Sg`, `App`, `Srv`, and `Cs` as appropriate. For example, `safCompType` for component types, `safSuType` for service unit types.

The naming of AMF entity types does not determine any hierarchical relationship among these types. Any such relationship is defined by other attributes. This allows for the reuse of the same type in multiple compound entity types, for instance, the same component type can be used in different service unit types, if required.

The structure of LDAP names is used instead to reflect the association of versioned entity types with a particular base entity type, as their DN is defined as an RDN followed by the base entity type's DN.

The RDN of an AMF versioned entity type is

```
safVersion=<version>
```

The `<version>` part of a name is not specified in more details. It is an `SaStringT`. Future releases of the Software Management Framework may define versioning rules.

The DN of an AMF versioned entity type is

`safVersion=<version>, safXXXType=<name>`

The upgrade campaign specification [9] allows for the specification of base and versioned entity types to be added to the system information model.

7.2.2 Other Attributes

The AMF entity types serve a different purpose at software delivery and in a running system. At delivery, they are partially specified entity prototypes, from which fully specified entity types need to be derived for runtime.

At software delivery, the most important role of the entity prototypes is to specify any restriction or limitation a software implementation may have. Based on this information, a correct deployment configuration can be determined. Therefore, this description should indicate the widest possible usage of the implementation delivered by a vendor, and it should typically contain only implementation limitations. This information is given as entity prototypes in the entity types XML file accompanying a software bundle and consists typically of attributes that provide compatibility information among types and attributes, which specify value ranges valid for the given implementation.

In a running system, though the information provided by the prototypes about the valid value ranges is still relevant for writable configuration attributes, it is not enough. Additional information is necessary to fully specify the types, as the AMF entity types are used to simplify the Availability Management Framework configuration by:

- collecting attributes that are common for all entities of a given type and specifying their value and by
- specifying the default values for those attributes that may be configured individually for each entity.

All these categories of attributes must be present and specified for the software catalog portion of the information model, so that the deployment configuration is fully specified. As opposed to this, the XML schema for the entity types file contains few mandatory elements and attributes of those elements. In the following sections discussing the XML schema of the AMF entity prototypes, it is always indicated whether an element or attribute is mandatory or not, particularly if it cannot be determined from the schema itself.

7.2.3 XML Schema for AMF Entity Prototypes

The complete XML schema is provided in [8]. This section presents only the elements describing the different AMF entity prototypes to provide a context for their usage for

anyone that intends to develop such prototypes and deliver them to SA Forum systems.

A versioned entity type is associated with a given software implementation or code. In case of the AMF versioned entity types, it is the component type and the component service type that are directly associated with a software implementation delivered in a bundle. Therefore, a software bundle must specify at least the component prototypes and the component service prototypes it delivers, which is done by providing an entity types file that describes these AMF entity prototypes. All the other AMF entity prototypes are optional in the entity types file, and their aim is to facilitate the task of the system integrator.

7.2.3.1 Component Prototype

The component prototype element (`CompType`) is mandatory in an entity types file accompanying a software bundle. Each component prototype element must have a name and a version attribute. It has the following mandatory elements:

- provided CS prototypes (`providesCSType`),
- component category, which is a choice of SA-aware (`saAware`), proxied (`proxied`), or non-proxied non-SA-aware (`unproxiedNonSaAware`), and a
- reference to the software bundle (`bundleReference`).

Optionally, it may also specify:

- the list of component prototypes with which this prototype is capable of collaborating in a redundancy scheme (`peerCompatibility`),
- whether the component prototype disallows restart (`disableRestart`),
- the error recovery action recommended by the vendor (`recoveryOnError`),
- a default CLC-CLI timeout (`defaultClcCliTimeOut`),
- the pair of CLC-CLI commands to start and stop active monitoring (`amStartCmd` and `amStopCmd`), and
- whether the component prototype is upgrade aware (`upgradeAware`).

7.2.3.1.1 Provided CS Prototypes

For each provided CS prototype, the name and the version attributes and the component capability must be specified. For any of the CS prototypes that indicate container role for the component prototype, the `isContainer` attribute needs to be set. The component capability is a choice of one of the following:

- x active and y standby (`xactiveandystandby`),
- x active or y standby (`xactiveorystandby`),

- one active or y standby (`oneactiveorystandby`),
- one active or one standby (`oneactiveoronestandby`),
- x active (`xactive`), or
- one active (`oneactive`).

For each x and y, an optional element is provided to specify any upper limit of the number of CSIs a component of the given prototype can take in the appropriate role; optionally, a default value may be given if there is a default value recommended by the vendor.

If the component prototype relies on another component prototype in providing the CS prototype, the optional `requiredCompType` element shall be used. This can be further refined by the `withCSType` element if a particular CS prototype needs to be provided by this other component prototype. The specification of these elements also means that components of the required prototype need to be collocated in the same service unit.

7.2.3.1.2 Component Category

Each component prototype belongs to a given component category. The elements that can be specified for each category are different:

⇒ `saAware`

- The `saAware` element describes component categories that implement the AMF API. Contained component prototypes require a container component prototype, which is described by the `containerCompType` element. Other categories are independent as they are directly managed by AMF. There is a mandatory choice between the `independent` and the `containerCompType` elements:

  ⇒ The `independent` element—it has

  - two mandatory child elements with the instantiate (`instantiateCmd`) and the cleanup (`cleanupCmd`) CLC-CLI commands, and
  - one optional child element:
    - proxied component prototypes (`proxiedCompType`) of components that can be proxied by components (of the given prototype) acting as proxy. These proxied component prototypes are specified by the name and version attributes and by the name and version of the corresponding component service prototype. In addition, for each proxied component prototype, the `healthcheck` prototypes that a proxy component (of the given prototype) recognizes and can implement may be specified.

⇒ The `containerCompType` element—it specifies the name and optionally the version of the container component prototype and the associated container CS prototype that a contained component prototype requires.

- Additional optional elements:
  - The `quiescingComplete` element specifies the time period within which a component of the given prototype is expected to answer with an `saAmfCSIQuiescingComplete()` call. It needs to be specified if the implementation requires a minimum timeout value for correct operation, in which case a default value can also be recommended by the vendor.
  - The `healthcheck` prototypes implemented by components of the given component prototype. Each of them has a mandatory key attribute and an optional healthcheck variant attribute. Additionally, the timers for health check period and maximum duration may be specified if they require a minimum value or if there is a vendor-recommended default.

⇒ `proxied`

- An optional element with the cleanup (`cleanupCmd`) CLC-CLI command that is required if a component of this prototype is intended to be instantiated as a local component.
- An optional attribute that specifies if the proxied component prototype is pre-instantiable. If it is so, the optional `quiescingComplete` timer may be specified in the same manner as it is used for `saAware` component prototypes.

⇒ `unproxiedNonSaAware`

- Mandatory attributes that are used to provide the instantiate, the terminate, and the cleanup CLC-CLI commands (`instantiateCmd` and `terminateCmd`).

7.2.3.1.3 CLC-CLI Commands

All CLC-CLI commands are specified as two strings: The first string contains the CLI command (including a relative path) and the second one specifies any required argument.

7.2.3.1.4 Upgrade Awareness

The `upgradeAware` element specifies the parameters of the callbacks recognized by the component prototype. The upgrade campaign specification predefines a set of conditions (Section 6.3.3.2) under which the Software Management Framework calls back registered processes. In the callback itself, a label and a user-defined string are returned as parameters to the registered process, which together identify for the process the condition under which the callback is made.

The labels and the user-defined strings identifying predefined conditions for the component prototype are specified in the `initCallback`, `backupCallback`, `rollbackCallback`, and `commitCallback` elements. The user-defined string is optional, and multiple strings can be specified for the same label. For each user-defined string, the further restrictions are described in the `condition` string element. For each of the callbacks, a `timer` element may specify the minimum waiting time and the waiting time recommended by the vendor.

Within the `upgradeAware` element, the `otherCallback` element is used to specify additional callbacks that the component prototype is capable of interpreting. They are specified the same way as the callbacks for predefined conditions, except that the `condition` string element must fully describe the condition for such a callback, for example, whether the callback needs to be invoked on the first step of a rolling upgrade before locking the deactivation unit.

7.2.3.1.5 Software Bundle Reference

Each component prototype must refer to the software bundle that delivers it (`bundleReference`). The reference is provided by the unique name of the software bundle, as described in Section 3.2.1.3. The software bundle element (Section 7.1.1) must be provided in the same entity prototypes file regardless of whether the entity prototypes file is accompanying this given bundle or another one.

7.2.3.1.6 Schema Summary

The following table summarizes the discussed XML schema elements for component prototypes, namely their presence and their attributes. For each element, the parent element is given, and a cross-reference is provided if the element is used in conjunction with or as an alternative to other elements. The presence is specified as M—mandatory, O—optional or A—alternative.

Table 6 XML Schema Elements for Component Prototypes Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`CompType`	M	name	M		
2			version	M	1, 5	
3	`peerCompatibility`	O	name	M	1	
4			version	O	3, 23, 24, 38, 39	
5	`providesCSType`	M	name	M	1	
6	`xactiveandystandby`	A			5	6-11
7	`xactiveorystandby`	A			5	6-11
8	`oneactiveorystandby`	A			5	6-11
9	`oneactiveoronestandby`	A			5	6-11
10	`xactive`	A			5	6-11
11	`oneactive`	A			5	6-11
12	`numMaxActiveCsi`	O	upper-Bound	M	6, 7, 10	
13			default	O	12, 14, 16, 18, 19, 31, 32, 43	
14	`numMaxStandbyCsi`	O	upper-Bound	M	6, 7, 8	
15	`saAware`	A			1	15, 25, 27
16	`quiescingCompleteTimeout`	O	lower-Bound	M	15, 25	
17	`healthcheck`	O			15	

Table 6 XML Schema Elements for Component Prototypes Specification (Continued)

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
18	`period`	M	lower-Bound	M	17	
19	`maxDuration`	M	lower-Bound	M	17	
20	independent	O				
21	`instantiateCmd`	M			20, 27	
22	`cleanupCmd`	M			20	
23	`proxiedCompType`	O	name	M	20	
24	`containedCompType`	O	name	M	20	
25	`proxied`	A	prein-stantia-ble	O	1	15, 25, 27
26	`cleanupCmd`	O			25	
27	`unproxiedNonSaAware`	A			1	15, 25, 27
28	`terminateCmd`	M			27	
29	`disableRestrt`	O			1	
30	`recoveryOnError`	O			1	
31	`defaultClcCliTimeOut`	O	lower-Bound	M	1	
32	`defaultCallbackTimeOut`	O	lower-Bound	M	1	
33	`amStartCmd`	O			1	34
34	`amStopCmd`	O			1	33
35	`bundleReference`	M	name	M	1	
36	`command`	M			21, 22, 26, 28, 33,34	

Table 6 XML Schema Elements for Component Prototypes Specification (Continued)

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
37	`args`	M			21, 22, 26, 28, 33,34	
38	`requiredCompType`	O	name	M	5	
39	`withCSType`	O	name	M	38	
40	`upgradeAware`	O			1	
41	`initCallback`	O	label	M	40	
42	`condition`	O	string-Param	M	41, 44-47	
43	`timer`	O	lower-Bound	M	41, 44-47	
44	`backupCallback`	O	label	M	40	
45	`rollabckCallback`	O	label	M	40	
46	`commitCallback`	O	label	M	40	
47	`otherCallback`	O	label	M	40	

7.2.3.2 Component Service Prototype

The component service prototype element (`CSType`) is mandatory in an entity types file accompanying a software bundle. It is used in association with the component prototype element. Each component service prototype is defined by the mandatory name and version attributes.

The `CSType` element has only an optional element, `csAttribute`, which is used to specify the name of an attribute in component service instances. In addition, for each attribute name, the `valueRestriction` element may contain further information on the type and value range of their values. Since the Availability Management Framework passes the attribute values as strings in the CSI assignment, the `valueRestriction` element is primarily provided to facilitate the configuration task.

7.2.3.2.1 Schema Summary

The following table summarizes the XML schema elements for CS prototypes.

Table 7 XML Schema Elements for CS Prototypes Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`CSType`	M	name	M		
2			version	M		
3	`csAttribute`	O	name	M	1	
4	`valueRestriction`	O	type	O	3	
5			lower-Bound	O	4	
6			upper-Bound	O	4	
7			default	O	4	

7.2.3.3 Service Unit Prototype

The specification of any service unit prototypes in the entity types file is optional. They can be specified using the `SUType` element, which has two mandatory attributes, the name and the version.

For a service unit prototype, the contained component prototypes need to be specified by their name and version. Optionally, the number of component instances can be provided for each component prototype by specifying a value range. If no `numInstances` element is specified, a service unit of the given prototype may contain only one instance of that given component prototype. If the `numInstances` element is specified, the lower or the upper bound may still not be specified. The default value for the lower bound is 1 and for the upper bound is no limit.

In the service unit prototype, the provided service prototypes are also listed (`providesServiceType`) with their names and versions. Each of them may specify if any other service prototype is required to provide the service prototype (`requiredServiceType`). For required service prototypes, the name is a mandatory attribute, but the version is optional.

A service unit prototype may also specify the fail-over element. If it is present, all CSIs of a service unit of this prototype need to be failed over together when a component of the service unit fails. If the fail-over element is not present, only the CSIs of the failed component need to be failed over; the CSIs of healthy components can be switched over.

7.2.3.3.1 Schema Summary

The following table summarizes the XML schema element for SU prototypes.

Table 8 XML Schema Elements for SU Prototypes Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`SUType`	O	name	M		
2			version	M	1, 3, 6	
3	`componentType`	M	name	M	1	
4	`numInstances`	O	minValue	O	3	
5			maxValue	O	4	
6	`providesServiceType`	M	name	M	1	
7	`requiredServiceType`	O	name	M	6	
8			version	O	7	
9	`suFailOver`	O			1	

7.2.3.4 Service Group Prototype

The specification of any service group prototypes in the entity types file is optional. They can be specified using the `SGType` element, which has two mandatory attributes, the name and the version.

For a service group prototype, the valid service unit prototypes need to be specified by their name and version. Only these service unit prototypes can be used to build a service group of this prototype. A service group may be built of service units of different prototypes.

A service group prototype must also specify the redundancy model by choosing one of the following elements:

- 2N (`twoN`)
- N+M (`nPlusM`)
- N-way (`nWay`)
- N-way-active (`nWayActive`)
- No redundancy (`noRedundancy`)

The vendor may recommend the auto-repair and the auto-adjust options. The presence of the auto-repair and the auto-adjust options reflects that the vendor recommends that the Availability Management Framework initiates respectively auto-repairs and auto-adjustments within a service group of the given prototype.

Finally, the vendor may recommend default values for the timer and the maximum counter of the component and service unit probations.

7.2.3.4.1 Schema Summary

The following table summarizes the XML schema element for service group prototypes.

Table 9 XML Schema Elements for Service Group Prototypes Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`SGType`	O	name	M		
2			version	M	1	1, 6, 7
3	`suType`	M	name	M	1	
4	`redModel`	M			1	
5	`autoRepairOption`	O			1	
6	`compProbation`	O	period	O	1	
7	`suProbation`	O	period	O	1	
8			counter-Max	O	6, 7	
9	`autoAdjust`	O	period	O	1	

7.2.3.5 Service Prototype

The specification of any service prototypes in the entity types file is optional. They can be specified using the `ServiceType` element, which has two mandatory attributes, the name and the version.

For a service prototype, the contained component service prototypes need to be specified by their name and version. Optionally, the number of component service instances can be provided for each component service prototype by specifying a value range. If no number is specified for a component service prototype, a service instance of the given prototype may have only one CSI of the specified component service prototype.

7.2.3.5.1 Schema Summary

The following table summarizes the XML schema element for service prototypes.

Table 10 XML Schema Elements for Service Prototypes Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	`ServiceType`	O	name	M		
2			version	M	1, 3	
3	`csType`	M	name	M	1	
4	`numInstances`	O	minValue	O	3	
5			maxValue	O	4	

7.2.3.6 Application Prototype

The specification of any application prototypes in the entity types file is optional. They can be specified using the AppType element, which has two mandatory attributes, the name and the version.

For an application prototype, only the contained service group prototypes are specified by their name and version. Both attributes are mandatory.

7.2.3.6.1 Schema Summary

The following table summarizes the XML schema element for application prototypes.

Table 11 XML Schema Elements for Application Prototype Specification

Item	Element	Element Presence	Attribute	Attribute Presence	Parent Element	Cross-Reference
1	AppType	O	name	M		
2			version	M	1, 3	
3	sgType	M	name	M	1	

8 Software Management Framework API

The Software Management Framework exports an API for processes that need to be informed about the initiation and progress of upgrade campaigns in the system in order to synchronize some application-level actions with the upgrade process. Such a process represents and acts on behalf of an upgrade-aware entity. In order to do so, it must initialize the Software Management Framework library and register with the Software Management Framework.

Typically, upgrade-aware entities are management applications or manager processes within applications that are capable of interpreting and acting upon the information received from the Software Management Framework.

The Software Management Framework provides callbacks to registered processes according to the upgrade campaign specification. Registered processes should be able to interpret these callbacks and initiate (directly or indirectly) the execution of the necessary application-level operations. These operations may range from checkpointing of application-level data synchronously with the upgrade campaign to application-level verification that would be difficult to carry out in any other way. In this second case, any detected failure needs to be reported back to the Software Management Framework to trigger an appropriate recovery operation. Therefore, registered processes must respond to the callbacks with the results of the application-level operations associated with the upgrade campaign. The Software Management Framework must take these responses into account in its decision about the continuation of the upgrade campaign.

There is a time limit the Software Management Framework must wait for responses to callbacks. If all the responses have been received or the timer has expired, the campaign proceeds according to the received results and taking timeouts as a success.

When a process registers with the Software Management Framework, it needs to state its **scope of interest**. The interest is expressed as a set of filters that are applied to callbacks specified in the upgrade campaign specification. The process will receive any callback in any upgrade campaign that specifies a label that matches any of the filters specified by the process.

Once a process unregisters with the Software Management Framework, it stops receiving callbacks from the Framework that match the particular set of filters that was given at the registration of that particular scope of interest. The same is true if the handle that was used at the registration becomes finalized or otherwise invalidated.

If a process registers with the Software Management Framework while an upgrade campaign is already in progress, the Software Management Framework is responsible for providing subsequent callbacks only. To notify the process that a campaign is already in progress, the Software Management Framework also initiates any callback that indicates the campaign initiation and matches the newly registered scope of interest; however, any time limit specified for the callback is ignored at this time.

8.1 Include File and Library Name

The following statement containing declarations of data types and function prototypes must be included[1] in the source of an application using the Software Management Framework API:

```
#include <saSmf.h>
```

To use the Software Management Framework API, an application must be bound with the following library:

```
libSaSmf.so
```

8.2 Type Definitions

The Software Management Framework uses the types described in the following sections.

8.2.1 Handles Used by the Software Management Framework

```
typedef SaUint64T SaSmfHandleT;
```

This type is used for the handle that is supplied by the Software Management Framework to a process during initialization of the Software Management Framework library and that is used by the process when it invokes functions of the Software Management Framework API.

1. The file `saSmf.h` is packaged together with the AIS C header files.

8.2.2 SaSmfPhaseT

The Software Management Framework indicates the phase of the upgrade campaign in its callbacks to upgrade-aware entities using the SaSmfPhaseT enumeration.

```
typedef enum{
     SA_SMF_UPGRADE        = 1,
     SA_SMF_ROLLBACK       = 2
} SaSmfPhaseT;
```

The values for the SaSmfPhaseT enumeration are as follows:

- SA_SMF_UPGRADE - The upgrade campaign is in the upgrade phase. When an upgrade campaign is initiated, it starts in the upgrade phase and remains so until completion, or until a rollback or a fallback is initiated by an administrative operation.
- SA_SMF_ROLLBACK - The upgrade campaign is in the rollback phase. Once a rollback administrative operation was issued on an upgrade campaign, the upgrade campaign moves to the rollback phase and remains so until completion, or until a fallback operation is initiated by the administrator.

8.2.3 SaSmfUpgrMethodT

```
typedef enum{
     SA_SMF_ROLLING        = 1,
     SA_SMF_SINGLE_STEP    = 2
} SaSmfUpgrMethodT;
```

The values for the SaSmfPUpgrMethodT enumeration are as follows:

- SA_SMF_ROLLING - The upgrade procedure uses the rolling upgrade method, as described in Section 3.3.3.2.1.
- SA_SMF_SINGLE_STEP - The upgrade procedure uses the single-step method, as described in Section 3.3.3.2.2.

8.2.4 SaSmfOfflineCommandScopeT

```
typedef enum {
     SA_SMF_CMD_SCOPE_AMF_SU    = 1,
     SA_SMF_CMD_SCOPE_AMF_NODE  = 2,
     SA_SMF_CMD_SCOPE_CLM_NODE  = 3,
     SA_SMF_CMD_SCOPE_PLM_EE    = 4,
     SA_SMF_CMD_SCOPE_PLM_HE    = 5
} SaSmfOfflineCommandScopeT;
```

The values of the `SaSmfOfflineCommandScopeT` enumeration are used to indicate, in the entity types file, the minimum scope of disruption anticipated by the software vendor during the execution of the offline installation and of the offline uninstallation commands (see Section 7.1.1.2). This type is also used in the `SaSmfSwBundle` object class to indicate the actual scope of disruption associated with each of the offline commands (see FIGURE 8 on page 161).

8.2.5 Types for State Management

8.2.5.1 SaSmfCmpgStateT

```
typedef enum {
     SA_SMF_CMPG_INITIAL                        = 1,
     SA_SMF_CMPG_EXECUTING                      = 2,
     SA_SMF_CMPG_SUSPENDING_EXECUTION           = 3,
     SA_SMF_CMPG_EXECUTION_SUSPENDED            = 4,
     SA_SMF_CMPG_EXECUTION_COMPLETED            = 5,
     SA_SMF_CMPG_CAMPAIGN_COMMITTED             = 6,
     SA_SMF_CMPG_ERROR_DETECTED                 = 7,
     SA_SMF_CMPG_SUSPENDED_BY_ERROR_DETECTED    = 8,
     SA_SMF_CMPG_ERROR_DETECTED_IN_SUSPENDING   = 9,
     SA_SMF_CMPG_EXECUTION_FAILED               = 10,
     SA_SMF_CMPG_ROLLING_BACK                   = 11,
     SA_SMF_CMPG_SUSPENDING_ROLLBACK            = 12,
     SA_SMF_CMPG_ROLLBACK_SUSPENDED             = 13,
     SA_SMF_CMPG_ROLLBACK_COMPLETED             = 14,
     SA_SMF_CMPG_ROLLBACK_COMMITTED             = 15,
     SA_SMF_CMPG_ROLLBACK_FAILED                = 16
} SaSmfCmpgStateT;
```

This enum represents the states of an upgrade campaign, as defined in Section 5.3.

8.2.5.2 SaSmfProcStateT

```
typedef enum {
     SA_SMF_PROC_INITIAL                = 1,
     SA_SMF_PROC_EXECUTING              = 2,
     SA_SMF_PROC_SUSPENDED              = 3,
     SA_SMF_PROC_COMPLETED              = 4,
     SA_SMF_PROC_STEP_UNDONE            = 5,
     SA_SMF_PROC_FAILED                 = 6,
     SA_SMF_PROC_ROLLING_BACK           = 7,
     SA_SMF_PROC_ROLLBACK_SUSPENDED     = 8,
     SA_SMF_PROC_ROLLED_BACK            = 9,
     SA_SMF_PROC_ROLLBACK_FAILED        = 10
} SaSmfProcStateT;
```

This enum represents the states of an upgrade procedure, as defined in Section 5.2.

8.2.5.3 SaSmfStepStateT

```
typedef enum {
     SA_SMF_STEP_INITIAL            = 1,
     SA_SMF_STEP_EXECUTING          = 2,
     SA_SMF_STEP_UNDOING            = 3,
     SA_SMF_STEP_COMPLETED          = 4,
     SA_SMF_STEP_UNDONE             = 5,
     SA_SMF_STEP_FAILED             = 6,
     SA_SMF_STEP_ROLLING_BACK       = 7,
     SA_SMF_STEP_UNDOING_ROLLBACK   = 8,
     SA_SMF_STEP_ROLLED_BACK        = 9,
     SA_SMF_STEP_ROLLBACK_UNDONE    = 10,
     SA_SMF_STEP_ROLLBACK_FAILED    = 11
} SaSmfStepStateT;
```

This enum represents the states of an upgrade step, as defined in Section 5.1.

8.2.5.4 SaSmfStateT

```
typedef enum {
     SA_SMF_CAMPAIGN_STATE      = 1,
     SA_SMF_PROCEDURE_STATE     = 2,
     SA_SMF_STEP_STATE          = 3
} SaSmfStateT;
```

This enum differentiates the sets of states for upgrade campaigns, upgrade procedures, and upgrade steps, as defined in Chapter 5.

8.2.5.5 SaSmfEntityInfoT

```
typedef enum {
     SA_SMF_ENTITY_NAME         = 1
} SaSmfEntityInfoT;
```

The preceding enum is used in Software Management Framework alarms and notifications (refer to Section 11.3.1) to convey additional information elements in the "Additional Information" field associated with alarms and notifications.

8.2.6 SaSmfCallbackScopeIdT

```
typedef SaUint32T SaSmfCallbackScopeIdT;
```

The `SaSmfCallbackScopeIdT` type represents the type of an identifier for an `scopeId` used by a process on a particular handle (obtained during the initialization of an instance of the Software Management Framework library) to register a particular scope of interest for callbacks. This identifier is used to associate the callbacks to the process for the scope of interest `scopeId`. The name space of the `scopeId` is limited to that particular handle, and the process may reuse it with another handle obtained by another initialization of the Software Management Framework library.

8.2.7 SaSmfCallbackLabelT

```
typedef struct {
     SaSizeT labelSize;
     SaUint8T *label;
} SaSmfCallbackLabelT;
```

In the upgrade campaign specification, each callback is identified by a label of type `xs:string`, which is converted by the Software Management Framework into a label of type `SaSmfCallbackLabelT`. These labels are used to identify the stage of the campaign; thus, an upgrade-aware entity can use them to identify the appropriate reaction to the callback identified by the label.

A process registering on behalf of an upgrade-aware entity specifies at registration the set of labels in which it is interested (that is, the upgrade-aware entity is capable of interpreting them) by specifying a set of filters. Any callback that has a label matching any of these filters is within the scope of interest of the registered process and results in a callback to the process.

8.2.8 Label Filters

The Software Management Framework supports several different types of filters and pattern matching algorithms, as defined by the following enumeration type.

8.2.8.1 SaSmfLabelFilterTypeT

```
typedef enum {
     SA_SMF_PREFIX_FILTER       = 1,
     SA_SMF_SUFFIX_FILTER       = 2,
     SA_SMF_EXACT_FILTER        = 3,
     SA_SMF_PASS_ALL_FILTER     = 4
} SaSmfLabelFilterTypeT;
```

This enum represents the values of a filter type. The corresponding pattern matching algorithms are explained later in Table 12.

8.2.8.2 SaSmfLabelFilterT

```
typedef struct {
     SaSmfLabelFilterTypeT filterType;
     SaSmfCallbackLabelT filter;
} SaSmfLabelFilterT;
```

The label filter structure defines the filter type and the filter pattern to be applied on a callback label to determine whether there is a need to callback a given user process for the callback specified in the upgrade campaign.

8.2.8.3 SaSmfLabelFilterArrayT

```
typedef struct {
     SaSizeT filtersNumber;
     SaSmfLabelFilterT *filters;
} SaSmfLabelFilterArrayT;
```

The label filter array structure defines one or more filters. Filters are passed to the Software Management Framework by a process at registration by invoking the `saSmfCallbackScopeRegister()` function. The Software Management Framework does the filtering to decide whether a callback specified in an upgrade campaign is issued to the registered process for a given registration by matching the filters (contents and type) against the label of a specific callback. If the callback label matches

any of the filters specified by a process at registration, the Software Management Framework invokes the specified callback of that registered process.

Table 12 Matching Algorithm for Each Filter Type

Filter Type	Matching Algorithm
SA_SMF_PREFIX_FILTER	The entire filter must match the first `labelSize` characters of the callback label. Match example: Filter=”abcd”, Callback label=”abcdxyz” Match example: Filter=”abcd”, Callback label=”abcd” Match example: Filter=”XYz”, Callback label=”XYzaB” Non-Match example: Filter=”xyz”, Callback label=”abcdxyz” Non-Match example: Filter=”Xyz”, Callback label=”xyzab” Non-Match example: Filter=”xyz”, Callback label=”xy” (The entire filter does not match the first part of the label; only the first two characters match.)
SA_SMF_SUFFIX_FILTER	The entire filter must match the last `labelSize` characters of the callback label. Match example: Filter=”xyz”, Callback label=”abcdxyz” Match example: Filter=”abCd”, Callback label=”abCd” Non-Match example: Filter=”abcd”, Callback label=”abcdxyz” Non-Match example: Filter=”xyz”, Callback label=”yz” (The entire filter does not match the last part of the callback label; only the last two characters match.)
SA_SMF_EXACT_FILTER	The entire filter must exactly match the entire callback label. Match example: Filter=”abc”, Callback label=”abc” Non-Match example: Filter=”ab”, Callback label=”abc” (The entire filter does not match the entire event pattern.)
SA_SMF_PASS_ALL_FILTER	Always matches, regardless of the filter or the callback label. It can be used with the empty string as a filter.

8.2.9 SaSmfCallbacksT

```
typedef struct {
     SaSmfCampaignCallbackT saSmfCampaignCallback;
} SaSmfCallbacksT;
```

The `SaSmfCallbacksT` structure defines the various callback functions that the Software Management Framework may invoke on a process.

8.3 Library Life Cycle

8.3.1 saSmfInitialize()

Prototype

```
SaAisErrorT saSmfInitialize(
      SaSmfHandleT *smfHandle,
      const SaSmfCallbacksT *smfCallbacks,
      SaVersionT *version
);
```

Parameters

`smfHandle` - [`out`] A pointer to the handle which identifies this particular initialization of the Software Management Framework, and which is to be returned by the Software Management Framework. The `SaSmfHandleT` type is defined in Section 8.2.1.

`smfCallbacks` - [`in`] If `smfCallbacks` is set to NULL, no callbacks are registered; If `smfCallbacks` is not set to NULL, it is a pointer to an `SaSmfCallbacksT` structure which contains the callback functions of the process that the Software Management Framework may invoke. Only non-NULL callback functions in this structure will be registered. The `SaSmfCallbacksT` type is defined in Section 8.2.9.

`version` - [`in/out`] As an input parameter, `version` is a pointer to a structure containing the required Software Management Framework version. In this case, `minorVersion` is ignored and should be set to 0x00.
As an output parameter, `version` is a pointer to a structure containing the version actually supported by the Software Management Framework. The `SaVersionT` type is defined in [1].

Description

This function initializes the Software Management Framework for the invoking process and registers the various callback functions. This function must be invoked prior to the invocation of any other Software Management Framework API function. The handle pointed to by `smfHandle` is returned as the reference to this association between the process and the Software Management Framework. The process uses this handle in subsequent communication with the Software Management Framework.

The `smfCallbacks` parameter points to a structure containing the callbacks that the Software Management Framework can invoke.

If the implementation supports the version of the Software Management Framework API specified by the `releaseCode` and `majorVersion` fields of the structure pointed to by the `version` parameter, `SA_AIS_OK` is returned. In this case, the structure pointed to by the `version` parameter is set by this function to:

- `releaseCode` = required release code
- `majorVersion` = highest value of the major version that this implementation can support for the required `releaseCode`
- `minorVersion` = highest value of the minor version that this implementation can support for the required value of `releaseCode` and the returned value of `majorVersion`

If the preceding condition cannot be met, `SA_AIS_ERR_VERSION` is returned, and the structure pointed to by the `version` parameter is set to:

if (implementation supports the required `releaseCode`)

`releaseCode` = required `releaseCode`

else {

if (implementation supports `releaseCode` higher than the required `releaseCode`)

`releaseCode` = the lowest value of the supported release codes that is higher than the required `releaseCode`

else

`releaseCode` = the highest value of the supported release codes that is lower than the required `releaseCode`

}

`majorVersion` = highest value of the major versions that this implementation can support for the returned `releaseCode`

`minorVersion` = highest value of the minor versions that this implementation can support for the returned values of `releaseCode` and `majorVersion`

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Software Management Framework library or a process that is providing the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES -The system is out of required resources (other than memory).

SA_AIS_ERR_VERSION - The version provided in the structure to which the `version` parameter points is not compatible with the version of the Software Management Framework implementation.

See Also

`saSmfFinalize()`

8.3.2 saSmfSelectionObjectGet()

Prototype

```
SaAisErrorT saSmfSelectionObjectGet(
      SaSmfHandleT smfHandle,
      SaSelectionObjectT *selectionObject
);
```

Parameters

`smfHandle` - [in] The handle which was obtained by a previous invocation of the `saAmfInitialize()` function and which identifies this particular initialization of the Software Management Framework. The `SaSmfHandleT` type is defined in Section 8.2.1.

`selectionObject` - [out] A pointer to the operating system handle that the invoking process can use to detect pending callbacks. The `SaSelectionObjectT` type is defined in [1].

Description

This function returns the operating system handle associated with the handle `smfHandle`. The invoking process can use this operating system handle to detect pending callbacks, instead of repeatedly invoking the `saSmfDispatch()` function for this purpose.

In a POSIX environment, the operating system handle is a file descriptor that is used with the `poll()` or `select()` system calls to detect incoming callbacks.

The operating system handle returned by `saSmfSelectionObjectGet()` is valid until `saSmfFinalize()` is invoked on the same handle `smfHandle`.

Return Values

`SA_AIS_OK` - The function completed successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Software Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` -The system is out of required resources (other than memory).

See Also

`saSmfInitialize()`, `saSmfDispatch()`, `saSmfFinalize()`

8.3.3 saSmfDispatch()

Prototype

```
SaAisErrorT saSmfDispatch(
      SaSmfHandleT smfHandle,
      SaDispatchFlagsT dispatchFlags
);
```

Parameters

`smfHandle` - [in] The handle which was obtained by a previous invocation of the `saSmfInitialize()` function and which identifies this particular initialization of the Software Management Framework. The `SaSmfHandleT` type is defined in Section 8.2.1.

`dispatchFlags` - [in] Flags that specify the callback execution behavior of the `saSmfDispatch()` function, which have the values `SA_DISPATCH_ONE`, `SA_DISPATCH_ALL`, or `SA_DISPATCH_BLOCKING`, as defined for the `SaDispatchFlagsT` type in [1].

Description

In the context of the calling thread, this function invokes pending callbacks for the handle `smfHandle` in a way that is specified by the `dispatchFlags` parameter.

Return Values

`SA_AIS_OK` - The function completed successfully. This value is also returned if this function is being invoked with `dispatchFlags` set to `SA_DISPATCH_ALL` or `SA_DISPATCH_BLOCKING`, and the handle `smfHandle` has been finalized.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - The `dispatchFlags` parameter is invalid.

See Also

`saSmfInitialize()`, `saSmfSelectionObjectGet()`, `saSmfFinalize()`

8.3.4 saSmfFinalize()

Prototype

```
SaAisErrorT saSmfFinalize(
     SaSmfHandleT smfHandle
);
```

Parameters

`smfHandle` - [in] The handle which was obtained by a previous invocation of the `saSmfInitialize()` function and which identifies this particular initialization of the Software Management Framework. The `SaSmfHandleT` type is defined in Section 8.2.1.

Description

The `saSmfFinalize()` function closes the association represented by the `smfHandle` parameter between the invoking process and the Software Management Framework. The process must have invoked `saSmfInitialize()` before it invokes this function. A process must call this function once for each handle it acquired by invoking `saSmfInitialize()`.

If the `saSmfFinalize()` function completes successfully, it releases all resources acquired when `saSmfInitialize()` was called. Moreover, it unregisters all entities registered for the particular handle. Furthermore, it cancels all pending callbacks related to the particular handle. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

If a process terminates, the Software Management Framework implicitly finalizes all instances of the Software Management Framework that are associated with the process, as described in the preceding paragraph.

After `saSmfFinalize()` completes successfully, the handle `smfHandle` and the selection object associated with it are no longer valid.

Return Values

`SA_AIS_OK` - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

See Also

`saSmfInitialize()`

8.4 Registration and Unregistration of the Scope of Interest

A process uses the following functions to register and unregister with the Software Management Framework its scope of interest for which the process wants to receive callbacks.

8.4.1 saSmfCallbackScopeRegister()

Prototype

```
SaAisErrorT saSmfCallbackScopeRegister(
      SaSmfHandleT smfHandle,
      SaSmfCallbackScopeIdT scopeId,
      const SaSmfLabelFilterArrayT *scopeOfInterest
);
```

Parameters

`smfHandle` - [in] The handle which was obtained by a previous invocation of the `saSmfInitialize()` function and which identifies this particular initialization of the Software Management Framework. The Software Management Framework must maintain the list of entities registered with each such handle. The `SaSmfHandleT` type is defined in Section 8.2.1.

`scopeId` - [in] An identifier that uniquely identifies the scope of interest registered by a process. The `SaSmfCallbackScopeIdT` type is defined in Section 8.2.6.

`scopeOfInterest` - [in] A pointer to a structure which specifies an array of filters to be used to select the callbacks in which the process is interested. The `SaSmfLabelFilterArrayT` type is defined in Section 8.2.8.3.

Description

The `saSmfCallbackScopeRegister()` function is used by a process to register its scope of interest with the Software Management Framework.

A process calls `saSmfCallbackScopeRegister()` to inform the Software Management Framework that the process would like to receive callbacks that have labels matching any of the filters specified by the structure to which the `scopeOfInterest` parameter points. The registered process must have supplied in its `saSmfInitialize()` call the appropriate set of callback functions (Section 8.2.9).

If a handle `smfHandle` of a process is finalized, all registrations made using this `smfHandle` will become implicitly unregistered, and any resources allocated for the registrations will be released.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INIT` - The previous initialization with `saSmfInitialize()` was incomplete, because the `saSmfCallbacks` pointer was NULL or the `saSmfCampaignCallback` element of the `SaSmfCallbacksT` structure was NULL.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Software Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` -The system is out of required resources (other than memory).

SA_AIS_ERR_EXIST - This value is returned if the scopeId already exists for this particular handle.

See Also

saSmfCallbackScopeUnregister(), SaSmfCampaignCallbackT, saSmfInitialize(), saSmfFinalize()

8.4.2 saSmfCallbackScopeUnregister()

Prototype

```
SaAisErrorT saSmfCallbackScopeUnregister(
      SaSmfHandleT smfHandle,
      SaSmfCallbackScopeIdT scopeId
);
```

Parameters

smfHandle - [in] The handle which was obtained by a previous invocation of the saSmfInitialize() function and which identifies this particular initialization of the Software Management Framework. The SaSmfHandleT type is defined in Section 8.2.1.

scopeId - [in] An identifier that uniquely identifies the scope of interest registered by a process. The SaSmfCallbackScopeIdT type is defined in Section 8.2.6.

Description

The saSmfCallbackScopeUnregister() function can be used by a process to unregister a previously registered scope of interest. As a consequence, the Software Management Framework will stop providing callbacks matching that particular scope of interest. During its life cycle, a process can register or unregister multiple times multiple scopes of interest.

The handle smfHandle in the saSmfCallbackScopeUnregister() call must be the same as that used in the corresponding saSmfCallbackScopeRegister() call.

Return Values

SA_AIS_OK - The function returned successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Software Management Framework library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES -The system is out of required resources (other than memory).

SA_AIS_ERR_NOT_EXIST - The scope of interest, identified by `scopeId` has not been registered previously for this handle.

See Also

`saSmfCallbackScopeRegister()`, `saSmfInitialize()`, `saSmfFinalize()`

8.5 Upgrade Campaign Progress Signaling and Response

8.5.1 SaSmfCampaignCallbackT

Prototype

```
typedef void (*SaSmfCampaignCallbackT) (
      SaSmfHandleT smfHandle,
      SaInvocationT invocation,
      SaSmfCallbackScopeIdT scopeId,
      const SaNameT *objectName,
      SaSmfPhaseT phase,
      const SaSmfCallbackLabelT *callbackLabel,
      const SaStringT params
);
```

Parameters

`smfHandle` - [in] The handle which was obtained by a previous invocation of the `saSmfInitialize()` function and which identifies this particular initialization of the

Software Management Framework. The SaSmfHandleT type is defined in Section 8.2.1.

invocation - [in] Used to match this invocation of SaSmfCampaignCallbackT with the corresponding invocation of saSmfResponse(). The SaInvocationT type is defined in [1].

scopeId - [in] An identifier that uniquely identifies the scope of interest registered by a process. The SaSmfCallbackScopeIdT type is defined in Section 8.2.6.

objectName - [in] A pointer to the DN of the object representing the upgrade campaign, procedure, or step within which the callback is initiated. The SaNameT type is defined in [1].

phase - [in] Indicates whether the given step is being executed as part of the upgrade phase, or whether the campaign is rolling back. The SaSmfPhaseT type is defined in Section 8.2.2.

callbackLabel - [in] Points to the structure containing the label specified for the custom callback in the upgrade campaign specification. The SaSmfCallbackLabelT type is defined in Section 8.2.7.

params - [in] The upgrade campaign specification may also indicate a formatted string to be passed to upgrade-aware entities. The SaStringT type is defined in [1].

Description

This callback is invoked when, within the upgrade campaign, a custom action is reached that specifies a callback operation. The parameter objectName points to the name of the context (that is, the upgrade campaign, procedure, or step) that best qualifies the scope within which the custom action was triggered.

Within the preceding context, callbackLabel points to a structure that represents a label in the upgrade campaign. This label identifies which particular custom action triggered the callback and shall be specified in such a way that the targeted upgrade-aware entities can easily recognize and interpret the callback (see Section 7.2.3.1.4). It is recommended that this label identifies at least the targeted upgrade-aware entities as well as the operation required from them. Additional parameters may be specified in the upgrade campaign specification as a formatted string.

The process shall indicate the result of the operation in a call to saSmfResponse(). Those upgrade-aware entities that cannot interpret the callback shall indicate this with the SA_ERR_NOT_SUPPORTED error code as soon as possible.

The Software Management Framework waits for all the responses, but not longer than the callback timeout period indicated in the upgrade campaign specification. If no waiting period is indicated, the Software Management Framework proceeds with the campaign immediately after having invoked the callbacks.

If an error is reported by any of the upgrade-aware entities by returning the `SA_AIS_ERR_FAILED_OPERATION` error code in a call to `saSmfResponse()`, the Software Management Framework shall interpret it as a failure of the custom action. Any timeout is interpreted as successful completion of the callback.

See Also

`saSmfCallbackScopeRegister()`, `saSmfCallbackScopeUnregister()`, `saSmfResponse()`

8.5.2 saSmfResponse()

Prototype

```
SaAisErrorT saSmfResponse(
      SaSmfHandleT smfHandle,
      SaInvocationT invocation,
      SaAisErrorT error
);
```

Parameters

`smfHandle` - [`in`] The handle which was obtained by a previous invocation of the `saSmfInitialize()` function and which identifies this particular initialization of the Software Management Framework. The `SaSmfHandleT` type is defined in Section 8.2.1.

`invocation` - [`in`] This parameter associates an invocation of this response function with a particular invocation of a callback function by the Software Management Framework. The `SaInvocationT` type is defined in [1].

`error` - [`in`] The response of the process to the associated callback. The value `SA_AIS_OK` is returned if the associated callback was successfully executed by the process; otherwise, an appropriate error as described in the corresponding callback must be returned (see Section 8.5.1). The `SaAisErrorT` type is defined in [1].

Description

The process responds to the Software Management Framework with the result of the execution of an operation that was initiated by the Software Management Framework when it invoked a callback specifying `invocation` to identify the initiated operation. In the `saSmfResponse()` call, the process gives that value of `invocation` back to the Software Management Framework, so that the Software Management Framework can associate this response with the callback request.

The process replies to the Software Management Framework when either (i) it cannot carry out the operations, or (ii) it has failed to successfully complete the execution of the operations, or (iii) it has successfully completed the operations.

This function may be called only by a process that registered the related scope of interest, that is, the `smfHandle` must be the same that was used when the process registered the scope of interest by invoking the `saSmfCallbackScopeRegister()` call.

Return Values

`SA_AIS_OK` - The function returned successfully.

`SA_AIS_ERR_LIBRARY` - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

`SA_AIS_ERR_TIMEOUT` - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The process may retry later.

`SA_AIS_ERR_BAD_HANDLE` - The handle `smfHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Availability Management Framework library or the provider of the service is out of memory and cannot provide the service.

`SA_AIS_ERR_NO_RESOURCES` -The system is out of required resources (other than memory).

See Also

`saSmfCallbackScopeRegister()`, `saSmfCallbackScopeUnregister()`, `saSmfInitialize()`, `saSmfFinalize()`, `SaSmfCampaignCallbackT`

9 Administrative API

9.1 Include File and Library Name

The appropriate Information Model Management Service header file and the Software Management Framework header file must be included in the source of an application using the Software Management Framework administrative API; for the name of the Information Model Management Service header file, see [4]. To use the Software Management Framework administrative API, an application must be bound to the Information Model Management Service library (for the library name, see [4]).

9.2 Type Definitions

The specification of Software Management Framework Administrative API requires the following types.

9.2.1 SaSmfAdminOperationIdT

```
typedef enum {
     SA_SMF_ADMIN_EXECUTE            = 1,
     SA_SMF_ADMIN_ROLLBACK           = 2,
     SA_SMF_ADMIN_SUSPEND            = 3,
     SA_SMF_ADMIN_COMMIT             = 4
} SaSmfAdminOperationIdT;
```

9.3 Software Management Framework Administrative API

The Software Management Framework administrative API shall be supported through the Information Model Management Service OM-API interface. The Information Model Management Service API `saImmOmAdminOperationInvoke_2()` or `saImmOmAdminOperationInvokeAsync_2()` functions (see [4]) shall be invoked with the appropriate `operationId` (see Section 9.2.1) and `objectName` to execute a particular administrative operation. In the following sections, the administrative API is described with the assumption that the Software Management Framework is an object implementer (runtime owner) for the various administrative operations that will be initiated as a consequence of invoking the `saImmOmAdminOperationInvoke_2()` or `saImmOmAdminOperationInvokeAsync_2()` functions with the appropriate `operationId` on the upgrade campaign object designated by `objectName`.

The API syntax for the administrative APIs shall use only the corresponding enumeration value for the `operationId` as explained in Section 9.2.1.

The return values explained in the sections below for various administrative operations shall be passed in the `operationReturnValue` parameter, which is provided by the invoker of the `saImmOmAdminOperationInvoke_2()` or `saImmOmAdminOperationInvokeAsync_2()` functions to obtain the return code from the object implementer, the Software Management Framework in this case.

9.3.1 SA_SMF_ADMIN_EXECUTE

Parameters

`operationId` = SA_SMF_ADMIN_EXECUTE

`objectName` - [in] pointer to the DN of the upgrade campaign object

`params` = NULL

Description

This administrative operation is invoked by an administrator to trigger the execution of an upgrade campaign represented by the object `objectName`. It can also be invoked to resume the execution of a upgrade campaign that was suspended in its forward path. Note that an upgrade campaign can be suspended either due to an invocation of an SA_SMF_ADMIN_SUSPEND operation, or due to an asynchronous failure detected by the Software Management Framework, or due to one or more upgrade steps of a campaign entering the `Undone` state. When an SA_SMF_ADMIN_EXECUTE operation is issued by calling `saImmOmAdminOperationInvoke_2()`, the function returns when one of the following events occurs:

- the Software Management Framework has successfully executed the upgrade campaign;
- the execution of the upgrade campaign is suspended through an invocation of an SA_SMF_ADMIN_SUSPEND operation;
- the upgrade campaign execution fails due to some error.

Similarly, when the operation is invoked by calling `saImmOmAdminOperationInvokeAsync_2()`, an `saImmOmAdminOperationCallback()` is invoked after one of the above events occurs.

The SA_SMF_ADMIN_EXECUTE operation can be issued only when the upgrade campaign is in the `Initial`, `Execution Suspended` or

Suspended by Error Detected states. The operation fails with an operationReturnValue of SA_AIS_ERR_BAD_OPERATION if the upgrade campaign execution or rollback is already in progress, that is, if the upgrade campaign is in a state other than Initial, Execution Suspended, or Suspended by Error Detected.

Software Management Framework implementations may choose to create all the procedure, step, and the other runtime objects necessary for the execution of the campaign when the execution is triggered. Alternatively, implementations may create the required runtime objects prior to invocation of the SA_SMF_ADMIN_EXECUTE operation on a campaign object.

Availability Management Framework configuration changes, such as deletion of existing nodes or service units, that are performed after an invocation of the SA_SMF_ADMIN_EXECUTE operation may cause the upgrade campaign to fail. Therefore, Availability Management Framework configuration changes during the execution of an upgrade campaign must be avoided.

Return Values

SA_AIS_OK - The execution of the upgrade campaign was performed successfully. The upgrade campaign is now in the Execution Completed state.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error must be returned in cases when the requested action is valid but not currently possible.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NO_MEMORY - The Software Management Framework is out of memory and, therefore, cannot provide service.

SA_AIS_ERR_NOT_EXIST - The upgrade campaign object identified by the name objectName does not exist.

SA_AIS_ERR_NO_OP - The upgrade campaign is already in the Executing state.

SA_AIS_ERR_BUSY - This error code is returned when a different upgrade campaign is already in progress, that is, it is in a state other than Initial, Campaign Committed, or Rollback Committed.

SA_AIS_ERR_BAD_OPERATION - The upgrade campaign is in a state other than Initial, Executing, Execution Suspended, or Suspended by Error Detected.

SA_AIS_ERR_ADMIN_SUSPENDING - The execution of the upgrade campaign was suspended through an invocation of an SA_SMF_ADMIN_SUSPEND operation. The upgrade campaign is now in the Suspending Execution state.

SA_AIS_ERR_ADMIN_ERROR_DETECTED - The upgrade campaign execution failed due to an upgrade procedure of the upgrade campaign notifying a failure caused by the step retry counter being exceeded during execution or caused by the detection of an asynchronous failure of an upgraded entity. The upgrade campaign is now in the Suspended by Error Detected state.

SA_AIS_ERR_ADMIN_FAILED - The upgrade campaign execution failed due to an upgrade procedure of the upgrade campaign notifying a failure caused by a reason other than either the step retry counter being exceeded during execution or the detection of an asynchronous failure. The upgrade campaign is now in the Execution Failed state.

9.3.2 SA_SMF_ADMIN_COMMIT

Parameters

operationId = SA_SMF_ADMIN_COMMIT

objectName - [in] pointer to the DN of the upgrade campaign object

params = NULL

Description

This administrative operation is invoked by an administrator to commit an upgrade campaign identified by objectName. It is also invoked by an administrator to commit an upgrade campaign rollback operation.

When a campaign execution or a campaign rollback has completed successfully (that is, it reached the Execution Completed state or the Rollback Completed state respectively), the administrator is expected to verify the upgrade or the rollback and commit the upgrade campaign by invoking an SA_SMF_ADMIN_COMMIT operation. When an upgrade campaign is committed, if any appropriate callback is specified, the Software Management Framework invokes registered processes to release resources such as backups and checkpoints associated with the upgrade campaign.

A Software Management Framework implementation may choose to free up any resources that it may have allocated during the execution of the upgrade campaign, such as the runtime objects pertaining to the upgrade campaign. For more details, see Section 6.5.2.

Return Values

SA_AIS_OK - The function completed successfully. If the campaign was in the `Execution Completed` state prior to the invocation of the operation, it is now in the `Execution Committed` state. If the campaign was in the `Rollback Completed` state prior to the invocation of the operation, it is now in the `Rollback Committed` state.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error must be returned in cases when the requested action is valid but not currently possible.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NO_MEMORY - The Software Management Framework is out of memory and, therefore, cannot provide service.

SA_AIS_ERR_NOT_EXIST - The upgrade campaign object identified by the name `objectName` does not exist.

SA_AIS_ERR_BAD_OPERATION - The upgrade campaign is in a state other than `Execution Completed` or `Rollback Completed`.

9.3.3 SA_SMF_ADMIN_SUSPEND

Parameters

`operationId` = SA_SMF_ADMIN_SUSPEND

`objectName` - [in] pointer to the DN of the upgrade campaign object

`params` = NULL

Description

This administrative operation is invoked by an administrator to suspend an upgrade campaign whose execution or rollback is in progress. When this operation is invoked, the Software Management Framework completes the upgrade steps (of one or more upgrade procedures) currently being executed before returning an `operationReturnValue` to the invoker. If the operation is invoked by calling `saImmOmAdminOperationInvoke_2()`, the function returns with an `operationReturnValue` of SA_AIS_OK if the campaign has been successfully suspended. If invoked by calling `saImmOmAdminOperationInvokeAsync_2()`, an `saImmOmAdminOperationCallback()` is invoked with an

`operationReturnValue` of `SA_AIS_OK` if the campaign has been successfully suspended.

Subsequent to the invocation of this operation, the administrator is expected to invoke either an `SA_SMF_ADMIN_EXECUTE` or an `SA_SMF_ADMIN_ROLLBACK` operation.

This specification does not discuss the consequences when neither of the operations is invoked.

It is possible that failures are encountered in the steps currently being executed when this operation is invoked. Appropriate error codes as described below are returned in such cases to indicate to the administrator that the campaign failed before it could be suspended.

Return Values

`SA_AIS_OK` - The function completed successfully. The upgrade campaign will either be in the `Execution Suspended` or `Rollback Suspended` state, depending on whether the operation was invoked when the campaign was in the `Executing` or `Rolling Back` state.

`SA_AIS_ERR_TRY_AGAIN` - The service cannot be provided at this time. The client may retry later. This error must be returned in cases when the requested action is valid but not currently possible.

`SA_AIS_ERR_NO_RESOURCES` - There are insufficient resources (other than memory).

`SA_AIS_ERR_NO_MEMORY` - The Software Management Framework is out of memory and, therefore, cannot provide service.

`SA_AIS_ERR_NOT_EXIST` - The upgrade campaign object identified by the name `objectName` does not exist.

`SA_AIS_ERR_NO_OP` - The upgrade campaign is already in a suspended state, that is, it is either in the `Execution Suspended` or the `Rollback Suspended` states.

`SA_AIS_ERR_BAD_OPERATION` - The upgrade campaign is in a state other than `Executing`, `Rolling Back`, `Execution Suspended`, and `Rollback Suspended`.

`SA_AIS_ERR_ADMIN_ERROR_DETECTED` - This error code is returned if the campaign execution fails after the invocation of the `SA_SMF_ADMIN_SUSPEND` operation due to an upgrade procedure notifying a failure caused by the step retry counter being exceeded during execution or caused by the detection of an asynchronous failure. The upgrade campaign is now in the `Suspended by Error Detected` state.

SA_AIS_ERR_ADMIN_FAILED - This error code is returned if the campaign fails after the invocation of the SA_SMF_ADMIN_SUSPEND operation due to an upgrade procedure of the upgrade campaign notifying a failure caused by a reason other than either the step retry counter being exceeded during execution or the detection of an asynchronous failure. The campaign will either be in the Execution Failed or in the Rollback Failed state, depending on whether the operation was invoked when the campaign was in the Executing or the Rolling Back state.

9.3.4 SA_SMF_ADMIN_ROLLBACK

Parameters

operationId = SA_SMF_ADMIN_ROLLBACK

objectName - [in] pointer to the DN of the upgrade campaign object

params = NULL

Description

This administrative operation is invoked by an administrator to perform a rollback operation on an upgrade campaign that has either completed or was suspended either due to an invocation of the SA_SMF_ADMIN_SUSPEND operation or due to a non-fatal failure (the failure did not lead to the Execution Failed or the Rollback Failed state). If the operation is invoked by calling saImmOmAdminOperationInvoke_2(), the function returns with an operationReturnValue of SA_AIS_OK if the campaign rollback was performed successfully. If invoked by calling saImmOmAdminOperationInvokeAsync_2(), an saImmOmAdminOperationCallback() is invoked with an operationReturnValue of SA_AIS_OK if the campaign rollback was performed successfully.

Return Values

SA_AIS_OK - Rollback was triggered successfully. The campaign will be in the Rolling Back state.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The client may retry later. This error must be returned in cases when the requested action is valid but not currently possible.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NO_MEMORY - The Software Management Framework is out of memory and, therefore, cannot provide service.

SA_AIS_ERR_NOT_EXIST - The upgrade campaign object identified by the name objectName does not exist.

SA_AIS_ERR_NO_OP - The upgrade campaign is already in the Rolling Back state.

SA_AIS_ERR_BAD_OPERATION - The upgrade campaign is in a state other than Execution Completed, Execution Suspended, Rolling Back, Rollback Suspended, or Suspended by Error Detected.

SA_AIS_ERR_ADMIN_SUSPENDING - The rollback of the upgrade campaign was suspended through an invocation of an SA_SMF_ADMIN_SUSPEND operation. The upgrade campaign is now in the Suspending Rollback state.

SA_AIS_ERR_ADMIN_SUSPENDED - The rollback of the upgrade campaign was suspended due to an AMF asynchronous error notification pertaining to the upgraded entities. The upgrade campaign is now in the Rollback Suspended state.

SA_AIS_ERR_ADMIN_FAILED - This error code is returned if the campaign rollback fails due to an upgrade procedure or the upgrade campaign notifying a failure. The upgrade campaign is now in the Rollback Failed state.

10 SMF UML Information Model

10.1 Notes on the Conventions Used in UML Diagrams

General explanation of the conventions used in the UML diagrams, such as the use of constraints, default values, and the like are presented in [1].

10.2 DN Formats for Software Management Framework UML Classes

Table 13 provides the format of the various DNs used to name Software Management Framework objects of the SA Forum Information Model. One format is defined for each object class.

Table 13 DN Formats

Object Class	DN Format for Objects of that Class
`SaSmfSwBundle`	"`safSmfBundle=...,*,safApp=safSmfService`"
`SaSmfCampaign`	"`safSmfCampaign=...,safApp=safSmfService`"
`SaSmfProcedure`	"`safSmfProcedure=...,safSmfCampaign=...,` `safApp=safSmfService`"
`SaSmfStep`	"`safSmfStep=`<integer>`,` `safSmfProcedure=...,safSmfCampaign=...,` `safApp=safSmfService`"
`SaSmfDeactivationUnit`	"`safSmfDu=...,safSmfStep=`<integer>`,` `safSmfProcedure=...,safSmfCampaign=...,` `safApp=safSmfService`"
`SaSmfActivationUnit`	"`safSmfAu=...,safSmfStep=`<integer>`,` `safSmfProcedure=...,safSmfCampaign=...,` `safApp=safSmfService`"
`SaSmfImageNode`	"`safImageNode=...,[safSmfAu=...,`|`safSmfDu=...,]` `safSmfStep=`<integer>`,safSmfProcedure=...,` `safSmfCampaign=...,safApp=safSmfService`"

10.3 Software Catalog Classes

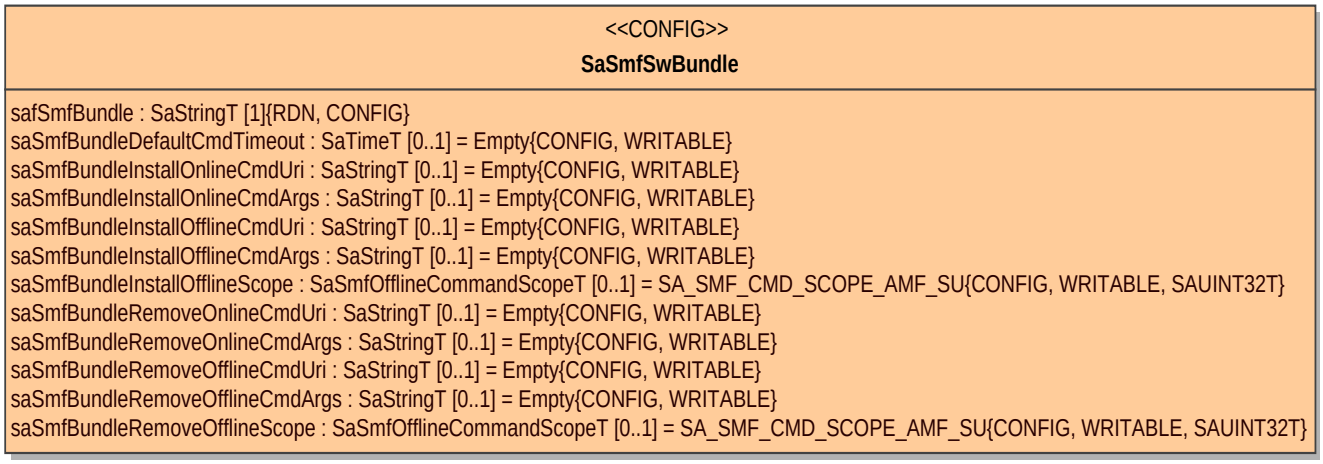
The following classes are specializations of the SaSmfBaseEntityType, SaSmfVersionedEntityType, and SaSmfSoftwareEntity object classes of the software catalog portion of the Software Management Framework Information Model (see FIGURE 2 in Section 3.2.1):

- SaAmfAppBaseType, SaAmfSGBaseType, SaAmfSUBaseType, SaAmfSvcBaseType, SaAmfCSBaseType, SaAmfCompBaseType - These configuration object classes are the specializations of the SaSmfBaseEntityType object class (presented in Section 3.2.1.2.1) for the Availability Management Framework, which also becomes the object implementer for objects of these classes. These classes are described in [2].
- SaAmfAppType, SaAmfSGType, SaAmfSUType, SaAmfSvcType, SaAmfCSType, SaAmfCompType - These configuration object classes are the specializations of the SaSmfVersionedEntityType object class (presented in Section 3.2.1.2.2) for the Availability Management Framework, which also becomes the object implementer for objects of these classes. These classes are described in [2].
- SaAmfApplication, SaAmfSG, SaAmfSU, SaAmfSI, SaAmfCSI, SaAmfComp - These configuration object classes are the specializations of the SaSmfSoftwareEntity object class (presented in Section 3.2.1.2.2) for the Availability Management Framework, which also becomes the object implementer for objects of these classes. These classes are described in [2].

The remaining object class of the software catalog portion of the Software Management Framework Information Model is:

- SaSmfSwBundle—This configuration object class defines the configuration attributes of a software bundle that has been delivered to the software repository associated with the SA Forum system. An object of this class can be created when the relevant software bundle is delivered to the software repository. It can be created later, but at the latest when an upgrade campaign requires this software bundle.
  For details, refer to Section 3.2.4 on page 32 and Section 6.3.1 on page 91.

FIGURE 8 SMF Bundle Class

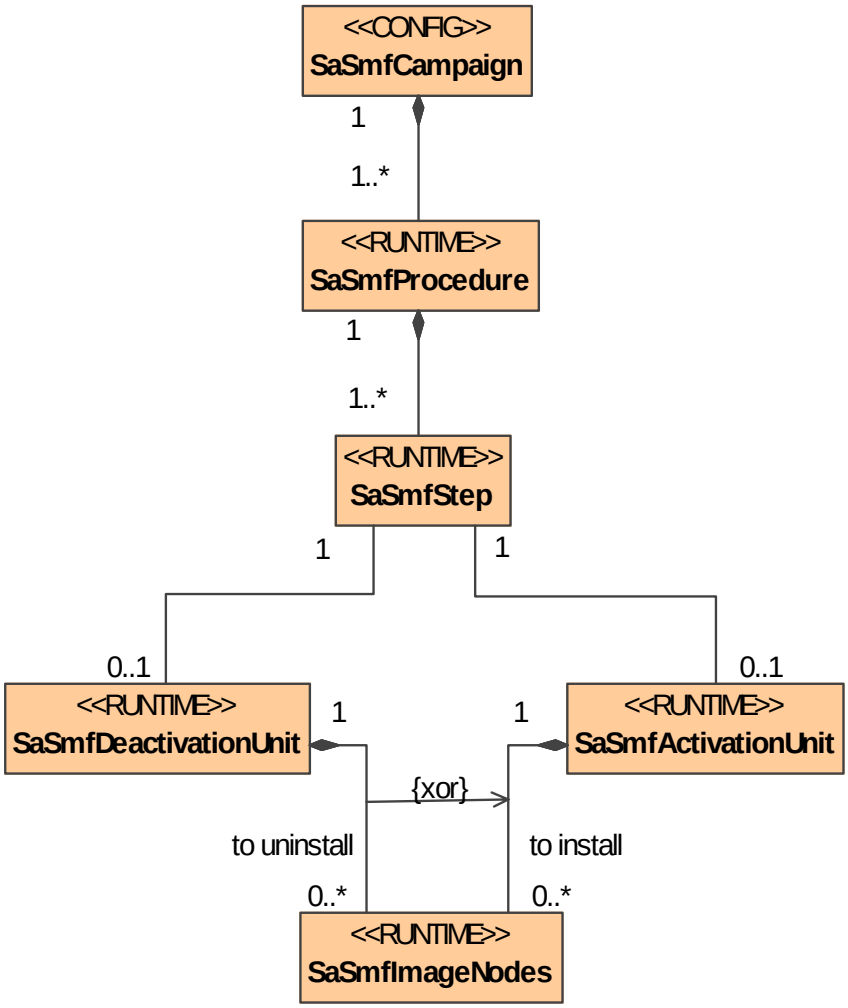


10.4 Upgrade Campaign Model Classes

10.4.1 Upgrade Campaign Model Overview

FIGURE 9 presents the relationships among the upgrade campaign model classes. For more details, refer to Section 3.3.1.1 on page 37.

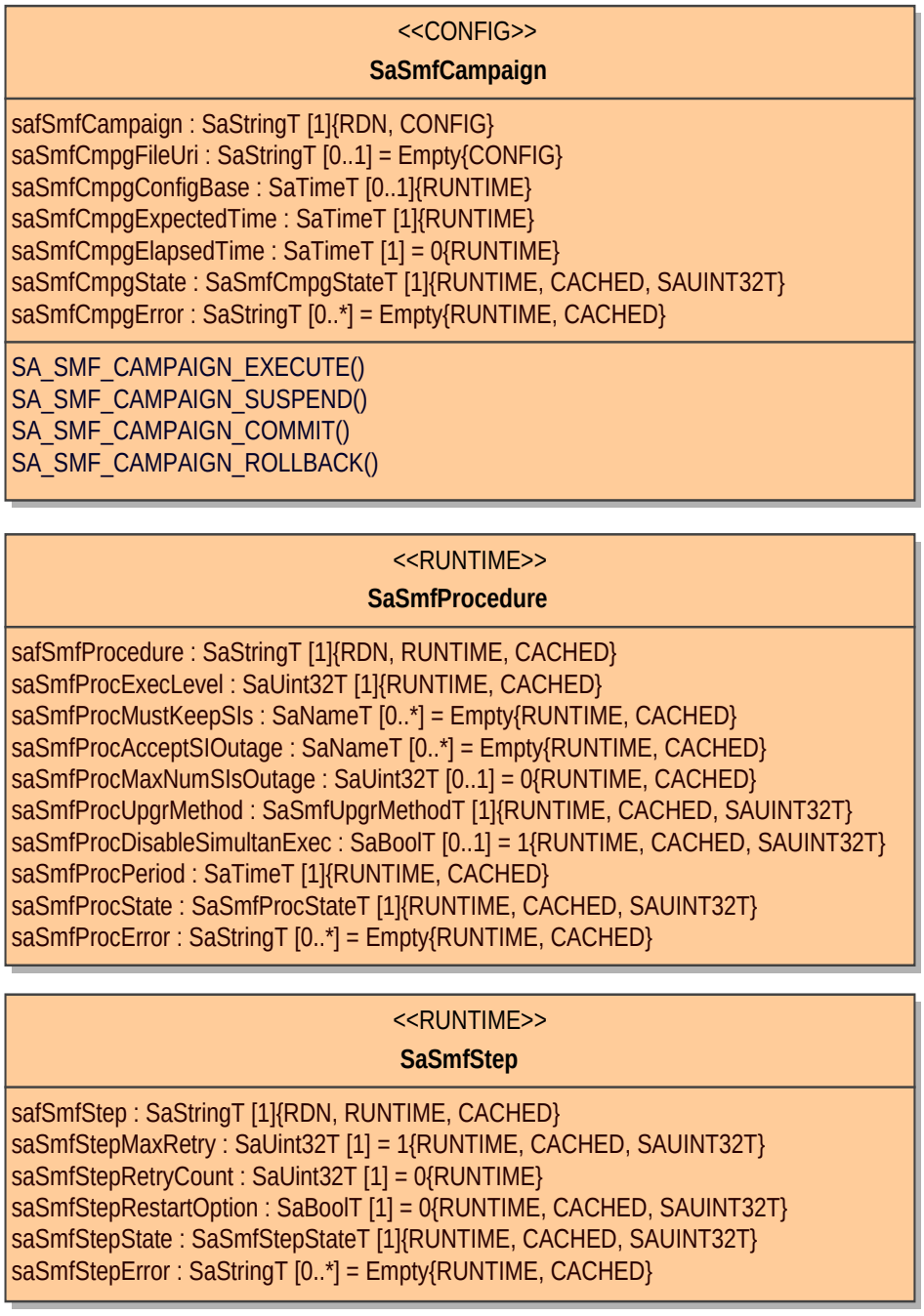
FIGURE 9 SMF Upgrade Campaign Status View



10.4.2 Upgrade Campaign, Upgrade Procedure, and Upgrade Step Classes

- `SaSmfCampaign`—This configuration object class defines the configuration and runtime attributes of an upgrade campaign and the administrative operations that can be applied to upgrade campaigns. The `saSmfCmpgFileUri` attribute indicates the location of the upgrade campaign specification file. It is the only attribute that needs to be provided at the creation of an object of this class. The Software Management Framework shall fetch the campaign from this location, and based on its contents fill the runtime attributes and create the runtime objects of the campaign as necessary.
  For details, refer to Section 3.3.1.1.1 on page 37.
- `SaSmfProcedure`—This runtime object class defines the runtime attributes of an upgrade procedure. For each upgrade procedure specified in the campaign, an object of this class is created by the Software Management Framework to reflect the status of the procedure execution during the campaign.
  For details, refer to Section 3.3.1.1.2 on page 38.
- `SaSmfStep`—This runtime object class defines the runtime attributes of an upgrade step that reflect the status of execution of the upgrade step during the campaign. For each specified upgrade procedure, the Software Management Framework calculates the number of upgrade steps required by the SA Forum system's current configuration, and for each of them creates a object of this class as a child of the appropriate procedure object.
  For details, refer to Section 3.3.1.1.3 on page 39.

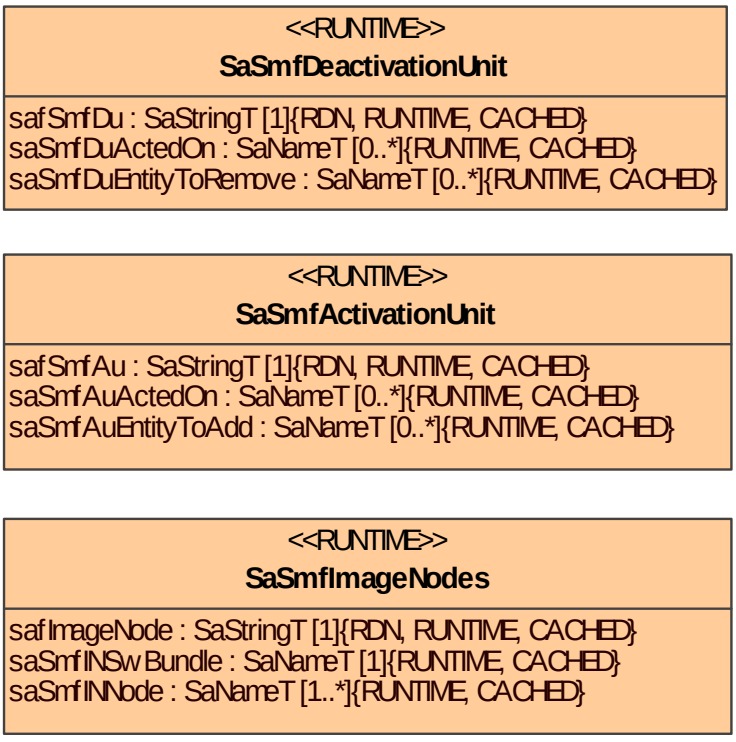
FIGURE 10 SMF Upgrade Campaign, Upgrade Procedure, and Upgrade Step Classes



10.4.3 SMF Deactivation Unit, Activation Unit, and Image-Nodes Classes

- `SaSmfDeactivationUnit`—This runtime object class defines the runtime attributes of a deactivation unit of an upgrade step. If the upgrade campaign does not explicitly specify the entities of the deactivation unit, the Software Management Framework must interpret the provided template and match it with the current system configuration to identify the list of entities in the deactivation unit of each step. For details, refer to Section 3.3.2.1 on page 40.
- `SaSmfActivationUnit`—This runtime object class defines the runtime attributes of an activation unit of an upgrade step. If the upgrade campaign does not explicitly specify the entities of the activation unit, the Software Management Framework must interpret the provided template and match it with the current system configuration to identify the list of entities in the activation unit of each step. For details, refer to Section 3.3.2.2 on page 41.
- `SaSmfImageNode`—This runtime object class defines the runtime attributes of a software bundle's association with a set of nodes on which it needs to be installed or uninstalled as part of an upgrade step.
  For details, refer to Section 3.3.2.1 on page 40.

FIGURE 11 SMF Deactivation Unit, Activation Unit, and Image-Nodes Classes



11 Alarms and Notifications

The Software Management Framework produces alarms and notifications to convey important information regarding the operational and functional state of the objects under its control to an administrator or a management system.

These reports vary in perceived severity and include alarms, which potentially require an operator intervention, and notifications that signify important state or object changes. A management entity should regard notifications, but they do not necessarily require an operator intervention.

The vehicle to be used for producing alarms and notifications is the Notification Service of the Service Availability™ Forum (abbreviated as NTF, see [3]). Hence, the various notifications are partitioned into categories as described in this service.

In some cases, this specification uses the term “Unspecified” for values of attributes that the vendor is at liberty to set to whatever makes sense in the vendor’s context, and the SA Forum has no specific recommendation regarding such values. Such values are generally optional from the CCITT Recommendation X.733 perspective (see [10]).

11.1 Setting Common Attributes

The following attributes of the notifications presented in Section 11.3 are not shown in their description, as the generic description presented here applies to all of them:

- Correlated Notifications - Correlation ids are supplied to correlate notifications that have been generated because of a related cause. By default, they should include the root and parent notifications to which the new notification relates. In case of alarms that are generated to clear certain conditions, that is, produced with a perceived severity of `SA_NTF_SEVERITY_CLEARED`, the correlation identifiers shall also include the notification identifier that the Notification Service assigned to the actual alarm notification when this alarm was generated.
- Event Time—The application might pass a timestamp or optionally pass an `SA_TIME_UNKNOWN` value, in which case the timestamp is provided by the Notification Service.
- Notification Id - Depending on the Notification Service function used to send the notification, this attribute is either implicitly set by the Notification Service or provided by the caller.
- Notifying Object - DN of the entity generating the notification. This name must conform to the SA Forum AIS naming convention and must contain at least the `safApp` RDN value portion of the DN set to the specified standard RDN value of

the SA Forum AIS Service generating the notification, that is, `safSmfService`. For details on the AIS naming convention, refer to [1].

The following notes apply to all Software Management Framework notifications presented in Section 11.3:

- Notification Class Identifier—The `vendorId` portion of the `SaNtfClassIdT` data structure must be set to `SA_NTF_VENDOR_ID_SAF` always, and the `majorId` field must be set to `SA_SVC_SMF` (as defined in the `SaServicesT` enumeration in [1]) for all notifications that follow the standard formats described in this specification. The `minorId` field will vary based on the specific notification.

11.2 Software Management Framework Alarms

The Software Management Framework does not issue any alarms at the time of publication of this specification.

11.3 Software Management Framework Notifications

The following sections describe a set of notifications that an Software Management Framework implementation shall produce.

11.3.1 Software Management Framework State Change Notifications

11.3.1.1 Upgrade Campaign State Change Notify

Description

The state of the upgrade campaign has changed due to administrative intervention or in the course of the execution of the upgrade campaign.

Table 14 Upgrade Campaign State Change Notify

NTF Attribute Name	Mandatory/Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the upgrade campaign whose state changed
Notification Class Identifier	NTF internal	`minorId` = 0x65 for Campaign
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_MANAGEMENT_OPERATION` or `SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_SMF_CAMPAIGN_STATE`
Old Attribute Value	Optional	Applicable value from enum `SaSmfCmpgStateT`
New Attribute Value	Mandatory	Applicable value from enum `SaSmfCmpgStateT`

11.3.1.2 Upgrade Procedure State Change Notify

Description

The state of an upgrade procedure has changed due to administrative intervention or in the course of the execution of the upgrade campaign.

Table 15 Upgrade Procedure State Change Notify

NTF Attribute Name	Mandatory/Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the upgrade procedure whose state changed
Notification Class Identifier	NTF-Internal	`minorid` = 0x66 for Procedure
Additional Text	Optional	Unspecified
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_SMF_PROCEDURE_STATE`
Old Attribute Value	Optional	Applicable value from enum `SaSmfProcStateT`
New Attribute Value	Mandatory	Applicable value from enum `SaSmfProcStateT`

11.3.1.3 Upgrade Step State Change Notify

Description

The state of an upgrade step has changed in the course of the execution of the upgrade campaign.

Table 16 Upgrade Step State Change Notify

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the upgrade step whose state changed
Notification Class Identifier	NTF-Internal	`minorid` = 0x67 for Step
Additional Text	Optional	Unspecified
Additional Information	Optional	`infoId` = `SA_SMF_ENTITY_NAME`, `infoType` = `SA_NTF_VALUE_LDAP_NAME`, `infoValue` = LDAP DN of the entity on which the upgrade step failed, if applicable
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_SMF_STEP_STATE`
Old Attribute Value	Optional	Applicable value from enum `SaSmfStepStateT`
New Attribute Value	Mandatory	Applicable value from enum `SaSmfStepStateT`

Index of Definitions

A
acceptable service outage 46
activation scope 43
activation units 39

B
base entity types 27

C
campaign builder 49
campaign *see* upgrade campaign

D
deactivation scope 43
deactivation units 39
deployment configuration 20
deployment phase 24
deployment *see* software deployment

E
entities
  software 27
  upgrade-aware 48
entity types file 33
execution level *see* procedure execution level
expected runtime outage 46

F
fallback 68

M
minimum service outage 46

O
offline installation and uninstallation operations 30
offline operations 30
online installation and uninstallation operations 30
online operations 30

P
procedure execution level 45
prototypes 28

R
registered processes 51
repository *see* software repository
rollback 65
rollforward 61
rolling upgrade 44

S
scope of disruption 108
scope of interest 127
service degradation 45
service outage 45
single-step upgrade 44
software bundle 28
software catalog 26
software delivery 23
software deployment 23
software entities 27
software entity type *see* types
software entity types 27
software installation 30
software repository 23
software uninstallation 30
symmetric activation units 42
symmetric upgrade scope 43

T
types
  base entity 27
  software entity 27
  versioned entity 28

U
upgrade campaign 34, 45
upgrade campaign period 47
upgrade method 43
  rolling 44
  single-step 44
upgrade procedure 43
upgrade procedure period 46
upgrade scope 43
upgrade step 39
upgrade-aware entities 48

V
version 28
versioned entity types 28