SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION

20 August 2015
Version: 4.1

Prepared by:

Joint Tactical Networking Center (JTNC)
33000 Nixie Way
San Diego, CA 92147-5110

Distribution Statement A - Approved for public release; distribution is unlimited (27 August 2015)
## REVISION SUMMARY

<table>
<thead>
<tr>
<th>Version</th>
<th>Revision Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next &lt;Draft&gt;</td>
<td>Initial Draft Release</td>
<td>30 November 2010</td>
</tr>
<tr>
<td>Next &lt;Draft&gt; 1.0.0.1</td>
<td>Applied SCA Next Errata Sheet v1.0</td>
<td>09 March 2011</td>
</tr>
<tr>
<td>Next &lt;Draft&gt; 1.0.0.2</td>
<td>Applied SCA Next Errata Sheet v2.0</td>
<td>14 September 2011</td>
</tr>
<tr>
<td>Candidate Release</td>
<td>Initial Release</td>
<td>27 December 2011</td>
</tr>
<tr>
<td>4.0</td>
<td>ICWG Approved Release</td>
<td>28 February 2012</td>
</tr>
<tr>
<td>4.0.1</td>
<td>Incorporated transition to JTNC and applied SCA 4.0 Errata Sheet v1.0</td>
<td>01 October 2012</td>
</tr>
<tr>
<td>4.1&lt;DRAFT&gt;</td>
<td>Naming Proposal Changes, Backwards Compatibility Changes, Scalable Components Changes, ScalableManagerChanges</td>
<td>31 December 2014</td>
</tr>
<tr>
<td>4.1</td>
<td>New :Process Collocation and Core Affinity Deployment Enhancement, Changes: Domain Late Registration, Allocation Properties, Domain Component Type Uniformity, Deployment Data, DomainManager and DeviceManager Instance Level Property Value, Clarification and Consistency edits</td>
<td>20 August 2015</td>
</tr>
<tr>
<td></td>
<td><strong>ICWG Approved</strong></td>
<td></td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

1 INTRODUCTION ............................................................................................................................................. 11  
1.1 Scope ...................................................................................................................................................... 11  
1.2 Document Conventions and Terminology ............................................................................................... 11  
  1.2.1 File and Directory Nomenclature ....................................................................................... 11  
  1.2.2 Requirements Language ............................................................................................................. 11  
  1.2.3 Core Framework Interface, Component and Operation Identification ..................................... 12  
1.3 Document Content .................................................................................................................................. 12  
1.4 Normative References ............................................................................................................................... 13  
1.5 Informative References .............................................................................................................................. 13  

2 OVERVIEW .................................................................................................................................................. 14  
2.1 Architecture Definition Methodology ...................................................................................................... 14  
  2.1.1 Component and Interface Definitions .................................................................................. 15  
  2.1.2 Component Implementation ..................................................................................................... 15  
2.2 Architecture Overview ............................................................................................................................ 16  
  2.2.1 System Architecture .................................................................................................................. 16  
  2.2.2 Application Architecture ............................................................................................................ 19  
    2.2.2.1 Reference Model ............................................................................................................... 20  
  2.2.3 Platform Devices and Services Architecture .............................................................................. 21  
  2.2.4 Core Framework Control Architecture ..................................................................................... 22  
  2.2.5 Structure ......................................................................................................................................... 22  
  2.2.6 Domain Profile ............................................................................................................................. 23  

3 SCA PLATFORM INDEPENDENT MODEL (PIM) ...................................................................................... 25  
3.1 Operating Environment ............................................................................................................................. 25  
  3.1.1 Operating System ......................................................................................................................... 26  
  3.1.2 Transfer Mechanism & Services .............................................................................................. 26  
    3.1.2.1 Log Service ......................................................................................................................... 26  
    3.1.2.2 Event Service and Standard Events .................................................................................. 27  
      3.1.2.2.1 Event Service .............................................................................................................. 27  
      3.1.2.2.2 StandardEvent Module ............................................................................................. 27  
    3.1.2.3 Types ..................................................................................................................................... 27  
      3.1.2.3.1 StateChangeCategoryType ....................................................................................... 27  
      3.1.2.3.2 StateChangeType ....................................................................................................... 27  

Distribution Statement on the Cover Page apply to all pages of this document.
3.1.2.3 Additional Services ........................................................................... 28
3.1.3 Core Framework ................................................................................. 29
  3.1.3.1 Common Elements ...................................................................... 30
    3.1.3.1.1 Interfaces ............................................................................. 30
      3.1.3.1.1.1 ComponentFactory ......................................................... 30
    3.1.3.1.2 Components ........................................................................ 32
      3.1.3.1.2.1 BaseComponent ............................................................. 32
      3.1.3.1.2.2 BaseFactoryComponent .............................................. 34
    3.1.3.1.3 Core Framework Base Types .............................................. 35
      3.1.3.1.3.1 DataTypes .................................................................... 35
      3.1.3.1.3.2 ObjectSequence ............................................................ 35
      3.1.3.1.3.3 FileException ................................................................. 35
      3.1.3.1.3.4 InvalidFileName ............................................................. 36
      3.1.3.1.3.5 InvalidObjectReference .............................................. 36
      3.1.3.1.3.6 InvalidProfile ................................................................. 36
      3.1.3.1.3.7 OctetSequence ............................................................... 36
      3.1.3.1.3.8 Properties ..................................................................... 36
      3.1.3.1.3.9 StringSequence ............................................................... 36
      3.1.3.1.3.10 UnknownProperties ................................................... 36
      3.1.3.1.3.11 DeviceAssignmentType ............................................. 36
      3.1.3.1.3.12 DeviceAssignmentSequence .................................... 36
      3.1.3.1.3.13 ErrorNumberType ....................................................... 37
      3.1.3.1.3.14 PortAccessType ........................................................... 37
      3.1.3.1.3.15 Ports ........................................................................... 37
      3.1.3.1.3.16 ComponentEnumType ................................................. 37
      3.1.3.1.3.17 ComponentType ........................................................... 38
      3.1.3.1.3.18 Components ................................................................. 39
      3.1.3.1.3.19 InvalidState ................................................................. 39
      3.1.3.1.3.20 PropertyActionType .................................................... 39
      3.1.3.1.3.21 PropertyType ............................................................... 39
      3.1.3.1.3.22 AllocationPropertyType .............................................. 40
      3.1.3.1.3.23 AllocationProperties ................................................... 40
      3.1.3.1.3.24 ManagerInfo ................................................................. 40
      3.1.3.1.3.25 SpecializedInfo Identifiers .......................................... 41
      3.1.3.1.3.26 UsesDeviceAssignmentType .................................... 41
      3.1.3.1.3.27 UsesDeviceAssignmentSeq ....................................... 41
  3.1.3.2 Base Application ...................................................................... 42
    3.1.3.2.1 Interfaces ........................................................................... 42
3.1.3.4.2.4 AggregateDeviceComponent

3.1.3.5 Framework Services

3.1.3.5.1 Interfaces

3.1.3.5.1.1 File

3.1.3.5.1.2 FileSystem

3.1.3.5.1.3 FileManager

3.1.3.5.2 Components

3.1.3.5.2.1 FileComponent

3.1.3.5.2.2 FileSystemComponent

3.1.3.5.2.3 FileManagerComponent

3.1.3.5.2.4 BasePlatformComponent

3.1.3.5.2.5 PlatformComponentFactoryComponent

3.1.3.5.2.6 ServiceComponent

3.1.3.5.2.7 ManageableServiceComponent

3.1.3.6 Domain Profile

3.1.3.6.1 Software Package Descriptor (SPD)

3.1.3.6.2 Software Component Descriptor (SCD)

3.1.3.6.3 Software Assembly Descriptor (SAD)

3.1.3.6.4 Properties Descriptor (PRF)

3.1.3.6.5 Device Package Descriptor (DPD)

3.1.3.6.6 Device Configuration Descriptor (DCD)

3.1.3.6.7 Domain Manager Configuration Descriptor (DMD)

3.1.3.6.8 Platform Deployment Descriptor (PDD)

3.1.3.6.9 Application Deployment Descriptor (ADD)

4 CONFORMANCE

4.1 Conformance Criteria

4.1.1 Conformance on the Part of an SCA Product

4.1.2 Conformance on the Part of an SCA OE component

4.2 Sample Conformance Statements
APPENDIX A: GLOSSARY

APPENDIX B: SCA APPLICATION ENVIRONMENT PROFILES

APPENDIX C: CORE FRAMEWORK INTERFACE DEFINITION LANGUAGE

APPENDIX D: PLATFORM SPECIFIC MODEL - DOMAIN PROFILE DESCRIPTOR FILES

APPENDIX E: MODEL DRIVEN SUPPORT TECHNOLOGIES

APPENDIX F: UNITS OF FUNCTIONALITY AND PROFILES
LIST OF FIGURES

Figure 2-1: Relationship between Component Definition and Implementation............... 15
Figure 2-2: Composition of a SCA System .............................................................. 16
Figure 2-3: SCA Component Hierarchy ................................................................. 18
Figure 2-4: Application Use of OE ........................................................................ 20
Figure 2-5: Conceptual Model of Application Components ....................................... 21
Figure 2-6: SCA Creation and Management Hierarchy ............................................ 22
Figure 2-7: Relationship of Domain Profile Descriptor File Types ............................... 25
Figure 3-1: Notional Relationship of OE and Application to an SCA AEP ...................... 26
Figure 3-2: Core Framework IDL Relationships ..................................................... 30
Figure 3-3: ComponentFactory Interface UML ....................................................... 31
Figure 3-4: BaseComponent UML ......................................................................... 32
Figure 3-5: BaseFactoryComponent UML .............................................................. 35
Figure 3-6: ComponentIdentifier Interface UML .................................................... 42
Figure 3-7: PortAccessor Interface UML ................................................................. 43
Figure 3-8: LifeCycle Interface UML ...................................................................... 46
Figure 3-9: TestableInterface Interface UML .......................................................... 47
Figure 3-10: PropertySet Interface UML ............................................................... 49
Figure 3-11: ControllableInterface Interface UML .................................................. 50
Figure 3-12: Application Component UML ............................................................ 52
Figure 3-13: ManageableApplicationComponent UML ......................................... 53
Figure 3-14: ApplicationControllerComponent UML .............................................. 54
Figure 3-15: ApplicationComponentFactoryComponent UML .............................. 55
Figure 3-16: AssemblyComponent UML ................................................................. 56
Figure 3-17: ApplicationManager Interface UML ................................................... 57
Figure 3-18: ApplicationManager Behavior ........................................................... 59
Figure 3-19: DeploymentAttributes Interface UML ............................................... 61
Figure 3-20: ApplicationFactory Interface UML ..................................................... 62
Figure 3-21: DomainManager Interface UML ......................................................... 66
Figure 3-22: DomainInstallation Interface UML ..................................................... 67
Figure 3-23: ComponentRegistry Interface UML .................................................... 70
Figure 3-24: FullComponentRegistry Interface UML ............................................. 72
Figure 3-25: EventChannelRegistry Interface UML ............................................... 73
Figure 3-26: ReleasableManager Interface UML .................................................... 75
Figure 3-27: ApplicationManagerComponent UML .............................................. 77
Figure 3-28: ApplicationFactoryComponent UML ................................................................. 79
Figure 3-29: ApplicationFactory Application Creation Behavior ......................................... 83
Figure 3-30: DomainManagerComponent UML ................................................................. 84
Figure 3-31: DeviceManagerComponent UML ..................................................................... 88
Figure 3-32: Device Manager Startup Scenario ..................................................................... 92
Figure 3-33: AdministratableInterface Interface UML .......................................................... 93
Figure 3-34: CapacityManagement Interface UML .............................................................. 94
Figure 3-35: State Transition Diagram for allocateCapacity and deallocateCapacity .......... 96
Figure 3-36: DeviceAttributes Interface UML ...................................................................... 97
Figure 3-37: AggregateDeviceAttributes Interface UML .................................................... 98
Figure 3-38: LoadableInterface Interface UML ................................................................... 99
Figure 3-39: ExecutableInterface Interface UML ............................................................... 101
Figure 3-40: AggregateDevice Interface UML .................................................................... 105
Figure 3-41: DeviceComponent UML .................................................................................. 106
Figure 3-42: State Transition Diagram for adminState ....................................................... 108
Figure 3-43: Release Child Device Scenario ....................................................................... 110
Figure 3-44: Release Parent Device Scenario ..................................................................... 111
Figure 3-45: LoadableDeviceComponent UML .................................................................... 112
Figure 3-46: ExecutableDeviceComponent UML ............................................................... 113
Figure 3-47: AggregateDeviceComponent UML .................................................................. 114
Figure 3-48: File Interface UML ........................................................................................ 115
Figure 3-49: FileSystem Interface UML .............................................................................. 118
Figure 3-50: FileManager Interface UML ............................................................................ 125
Figure 3-51: FileComponent UML ..................................................................................... 128
Figure 3-52: FileSystemComponent UML .......................................................................... 129
Figure 3-53: FileManagerComponent UML ....................................................................... 130
Figure 3-54: BasePlatformComponent UML ...................................................................... 131
Figure 3-55: PlatformComponentFactoryComponent UML ............................................. 132
Figure 3-56: ServiceComponent UML ............................................................................... 133
Figure 3-57: ManageableServiceComponent UML ............................................................ 134
FOREWORD

Introduction. The Software Communications Architecture (SCA) is published by the Joint Tactical Networking Center (JTNC). This architecture was developed to assist in the development of Software Defined Radio (SDR) communication systems, capturing the benefits of recent technology advances which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs. The architecture is also applicable to other embedded, distributed-computing applications such as Communications Terminals or Electronic Warfare (EW). The SCA has been structured to:

1. provide for portability of applications software between different SCA implementations,
2. leverage commercial standards to reduce development cost,
3. reduce software development time through the ability to reuse design modules, and
4. build on evolving commercial frameworks and architectures.

The SCA is deliberately designed to meet commercial application requirements as well as those of military applications. Since the SCA is intended to become a self-sustaining standard, a wide cross-section of industry has been invited to participate in the development and validation of the SCA. The SCA is not a system specification but an implementation independent set of rules that constrain the design of systems to achieve the objectives listed above.

Core Framework. The Core Framework (CF) defines the essential "core" set of open software interfaces and profiles that provide for the deployment, management, interconnection, and intercommunication of software application components in an embedded, distributed-computing communication system. In this sense, all interfaces defined in the SCA are part of the CF.
1 INTRODUCTION

The SCA establishes an implementation-independent framework with baseline requirements for the development of software for SDRs. The SCA is an architectural framework that was created to maximize portability, configurability of the software (including changing waveforms), and component interoperability while still allowing the flexibility to address domain specific requirements and restrictions. Constraints on software development imposed by the framework are on the interfaces and the structure of the software and not on the implementation of the functions that are performed. The framework places an emphasis on areas where reusability is affected and allows implementation unique requirements to determine a specific application of the architecture.

1.1 SCOPE

This document together with its appendices as specified in the Table of Contents provides a complete definition of the SCA.

The goal of this specification is to provide for the deployment, management, interconnection, and intercommunication of software components in embedded, distributed-computing communication systems.

1.2 DOCUMENT CONVENTIONS AND TERMINOLOGY

1.2.1 File and Directory Nomenclature

The terms "file" and "filename" as used in the SCA, refer to both a "plain file" (equivalent to a POSIX "regular file") and a directory. An explicit reference is made within the text when referring to only one of these.

Pathnames are used in accordance with the POSIX specification definition and may reference either a plain file or a directory. An "absolute pathname" is a pathname which starts with a "/" (forward slash) character - a "relative pathname" does not have the leading "/" character. A "path prefix" is a pathname which refers to a directory and thus does not include the name of a plain file.

1.2.2 Requirements Language

The word "shall" is used to indicate absolute requirements of this specification which must be strictly followed in order to achieve compliance. No deviations are permitted.

The phrase "shall not" is used to indicate a strict and absolute prohibition of this specification.

The word "should" is used to indicate a recommended course of action among several possible choices, without mentioning or excluding others. "Should not" is used to discourage a course of action without prohibiting it.

The word "may" is used to indicate an optional item or allowable course of action within the scope of the specification. A product which chooses not to implement the indicated item must be able to interoperate with one that does without impairment of required behavior.

The word "is" (or equivalently "are") used in conjunction with the association of a value to a data type indicates a required value or condition when multiple values or conditions are possible.
1.2.3 Core Framework Interface, Component and Operation Identification

References to interface names, their operations and defined XML elements/attributes within this specification are presented in italicized text. All interface names are capitalized. Interface attributes, operation parameters, and components are presented in plain text. "CF" precedes references to Core Framework Base Types (3.1.3.1.3)

1.3 DOCUMENT CONTENT

The Foreword and Section 1 Introduction, of this document provide an introduction to this specification and identifies the definitions and rules for its usage.

Section 2, Overview, provides an overview of the SCA as well as a description of the interfaces and behaviors prescribed by the specification.

Section 3, SCA Platform Independent Model (PIM), provides the detailed description of the architecture framework and the specification requirements.

Section 4, Conformance, defines the authorities, requirements and criteria for product certification in accordance with this specification.

Appendix A: Glossary, contains a glossary of terms and acronyms used in this specification.

Appendix B: SCA Application Environment Profiles, provides the specific requirements for the SCA Application Environment Profiles (AEP) required as part of compliance to this specification.

Appendix C: Core Framework Interface Definition Language, contains the IDL code used to define the interfaces required by this specification.

Appendix D: Platform Specific Model - Domain Profile Descriptor Files, provides a mapping of the SCA PIM to specific descriptor file representations as part of compliance to this specification.

Appendix D-1: Platform Specific Model – Document Type Definition Files, defines the SCA Domain Profiles using XML Document Type Definition (DTD) files.

Appendix D-1 Attachment 1: Common Properties Definitions, provides the common properties to be used for component definitions.

Appendix E: Model Driven Support Technologies, provides a mapping of the SCA PIM to specific platform transports and technologies as part of compliance to this specification.

Appendix E-1: Application Interface Definition Language Platform Independent Model Profiles, defines two Object Management Group (OMG) Interface Definition Language (IDL) profiles which enable Platform Independent Modeling of Software Defined Radio (SDR) applications. The two profiles are the Full PIM IDL Profile (Full) and the Ultra-Lightweight PIM IDL Profile (ULw).

Appendix E-2: Platform Specific Model - Common Object Request Broker Architecture, defines the platform specific transport and technology model using the Common Object Request Broker Architecture (CORBA).

Appendix E-2 Attachment 1: SCA CORBA Profiles (from CORBA/e), lists the Full/LW/ULW Profile features used from CORBA/e.
Appendix E-2 Attachment 2: SCA CORBA Profiles (from RT CORBA), lists the Full/LW/ULW Profile features used from RT CORBA.


Appendix F: Units of Functionality and Profiles, defines Units of Functionality (UOFs) and Profiles used to achieve scalable levels of conformance with this specification.

Appendix F Attachment 1 SCA Conformance Mappings, provides a list of the requirement text, corresponding requirement number, applicable UOF(s) and corresponding document section number where the requirement text can be found.

1.4 NORMATIVE REFERENCES

The following documents contain provisions or requirements which by reference constitute requirements of this specification. Applicable versions are as stated.


1.5 INFORMATIVE REFERENCES

The following is a list of documents referenced within this specification or used as reference or guidance material in its development.


2 OVERVIEW

This section presents an architectural overview of the SCA which defines the fundamental organization of the components that compose this specification. A high-level description of the interfaces and components, their responsibilities, as well as their relationship to each other and the environment are also provided. Technical details and specific requirements of the architecture and individual components are contained in section 3.

The goal of this specification is to provide for the deployment, management, interconnection, and intercommunication of software components in embedded, distributed-computing communication systems. This specification is targeted towards facilitating the development of SDRs with the additional goals of maximizing software application portability, reusability, and scalability through the use of commercial protocols and products.

Although there are many definitions of a SDR, it is in essence a radio or communication system whose output signal is determined by software. In this sense, the output is entirely reconfigurable at any given time, within the limits of the radio or system hardware capabilities (e.g. processing elements, power amplifiers, antennas, etc.) merely by loading new software as required by the user. Since this software determines the output signal of the system, it is typically referred to as "waveform software" or simply as the "waveform" itself. This ability to add, remove, or modify the output of the system through reconfigurable and re-deployable software, leads to communication systems capable of multiple mode operation (including variable signal formatting, data rates, and bandwidths) within a single hardware configuration. Simultaneous multi-mode operation is possible when a multi-channel configuration is available.

2.1 ARCHITECTURE DEFINITION METHODOLOGY

The architecture has been developed using an object-oriented approach including current best practices from software component models and software design patterns. Unless stated, no explicit grouping or separation of interfaces is required within an implementation. The interface definitions and required behaviors that follow in section 3, define the responsibilities, roles, and relationships of their component realizations.

The specification uses the Unified Modeling Language (UML) [5], [6], defined by the Object Management Group (OMG), to graphically represent interfaces, components, operational scenarios, use cases, and sequence diagrams; the OMG defined Interface Definition Language (IDL) [7] to provide the textual representation of the interfaces (see Appendix E-3 for the mapping); and eXtensible Markup Language (XML) [8] is used to create the SCA Domain Profile elements which identify the capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up an SCA-compliant system.

IDL fragments appear in section 3 to illustrate interfaces but the IDL in Appendix C takes precedence. The terms "Domain Profile" and "profile" are used to refer to either the raw XML format of these files as well as these same files in a parsed format. References to a specific Domain Profile file (e.g. Software Assembly Descriptor (SAD), Device Configuration Descriptor (DCD)) refer to the raw XML format per the definitions in section 3.1.3.6.
2.1.1 Component and Interface Definitions

The SCA specifies requirements using both interface and component definitions. An interface definition includes the formal operation signatures and associated behaviors. A component is "an autonomous unit within a system or subsystem" which has the following characteristics:

- Provides one or more Interfaces which users may access, and
- Hides its internals and makes them inaccessible other than as provided by its Interfaces.

The component definitions reference interface definitions (which may not be component-unique) and specify any required behaviors, constraints or associations that must be adhered to when their corresponding SCA products are built.

Within this specification components are defined as stereotypes that represent the bridge between the interface definitions and the products that will be built in accordance with the SCA.

2.1.2 Component Implementation

Component implementations must realize a component definition and satisfy all of its aggregated requirements as shown in Figure 2-1. The term "component" in this document will alternatively refer to a component definition or a concrete implementation depending on the context. Where the distinction is not obvious, the text will append "definition" or "implementation" as a modifier.

---

**Figure 2-1: Relationship between Component Definition and Implementation**
2.2 ARCHITECTURE OVERVIEW

Since the functionality of software itself is virtually limitless, there is a large dependency placed on the ability to select and configure the appropriate hardware to support the software available or required for a specific system. The selection of hardware is not restricted to the input/output (I/O) devices typically associated with communication systems (analog-to-digital converters, power amplifiers, etc.). It is dependent on the type and capabilities of the processing elements (General Purpose Processors (GPP), Digital Signal Processors (DSP), Field-Programmable Gate Arrays (FPGA), etc.) that are required to be present, since typically the software required to generate a given output signal will consist of many components of different types based on performance requirements. From an illustrative view, this results in a system that is represented by a variable collection of hardware elements which need to be connected together to form communication pathways based on the specific software loaded onto the system. The role of the SCA is then to provide a common infrastructure for managing the software and hardware elements present in a system and ensuring that their requirements and capabilities are commensurate. The SCA accomplishes this function by defining a set of interfaces that isolate the system applications from the underlying hardware. This set of interfaces is referred to as the Core Framework of the SCA.

Additionally, the SCA provides the infrastructure and support elements needed to ensure that once software components are deployed on a system, they are able to execute and communicate with the other hardware and software elements present in the system.

2.2.1 System Architecture

An SCA-based system consists of an Operating Environment (OE) and one or more Applications as shown in Figure 2-2.

![Figure 2-2: Composition of a SCA System](image-url)
The SCA differentiates between application, i.e. waveform, software that manipulates input data and determines the output of the system and OE software that provides the capabilities to host waveforms and allow them to access system resources. The software components that provide access to the system hardware resources are referred to as SCA devices, which implement the Base Device Interfaces. Non-hardware (software-only) components provided by the system for use by multiple applications are generically referred to as services; however the SCA does not specify an interface for these components.

The SCA standardizes a collection of component definitions, but does not place implementation requirements (e.g. transport mechanisms) on the realized software. A notional representation of the hierarchy of the significant SCA components is shown in Figure 2-3.
Figure 2-3: SCA Component Hierarchy
The OE provides the capability to manage and execute SCA components and consists of the Operating System, Transfer Mechanism, Core Framework Control and Platform Devices and Services.

The SCA includes real-time embedded operating system (RTOS) functions that provide multi-threaded support for all software executing on the system, including applications, devices, and services.

The SCA leverages transfer mechanisms to provide standardized client/server operations. Client/server communications may be co-located or distributed across different processors. The transfer mechanism structure may be comprised of object request semantics, transfer and message syntax, and transports.

The following sections describe the architectural structure of the constituent portions of an SCA-based system with the exception of the Operating System and Transfer Mechanism whose architecture is not specified by the SCA.

2.2.2 Application Architecture

SCA Applications (typically waveforms) contain the following components:

- **Base Application Components**: ApplicationComponent, ApplicationComponentFactoryComponent (optional), AssemblyComponent, ManageableApplicationComponent and ApplicationControllerComponent, which utilize BaseComponents and provide management of application software.

Application components realize the Base Application and Common interfaces that are described below:

- **Base Application Interfaces**: ComponentIdentifier, ControllableInterface, PortAccessor, Lifecycle, TestableInterface, and PropertySet;

- **Common Interfaces**: ComponentFactory.

The Base Application interfaces provide APIs for the control and configuration of software components. Application developers may extend these capabilities by creating specialized interfaces for the application. The design of an application component's internal functionality is not dictated by the SCA and is left to the application developer.

Applications interface with the other components of an SCA-based system as presented in Figure 2-4. An application consists of multiple software components, e.g. ManageableApplicationComponents, which are loaded onto the appropriate processing resource. These components may be managed by the Framework Control Components. ManageableApplicationComponents communicate with each other or with the services and devices provided by the system via extensions of the PortAccessor interface. It is intended that the APIs used by Platform Devices and Service Components be standardized for a given system or domain so that all communications to and from the application are uniform across multiple systems. However, the standardization of these interfaces is outside the scope of this specification since they are system and domain specific.

An application may access OS functionality but is restricted to the operations enumerated in the Appendix B which is a subset of the Portable Operating System Interface (POSIX) specification.
[9]. The specification contains multiple POSIX profiles to allow an implementation to tailor a product to a minimal set of POSIX features.

Figure 2-4: Application Use of OE

2.2.2.1 Reference Model

All application components are comprised of the Base Application interfaces and use devices. Specific interface compositions are designed in accordance with their functional requirements but that mapping is not identified by this specification.

Figure 2-5 shows example inheritance relationships for an application component. As illustrated, the developer can determine which optional interfaces are required for a specific component, denoted by the comment tag over the association line (e.g. CONNECTABLE) in the UML diagram. Each comment tag corresponds to a Unit of Functionality defined in Appendix F. The operations and attributes provided by the LifeCycle, TestableInterface, PortAccessor, ComponentIdentifier, ControllableInterface and PropertySet interfaces establish a common approach for interfacing with a component in an SCA environment. The PortAccessor interface is used for pushing or pulling messages between application components and devices. An application component may consist of zero or more input and output message ports. Figure 2-5 also shows an example of a more specialized component that could be realized to provide implementation specific functionality (Note: the component represents a generic example).
2.2.3 Platform Devices and Services Architecture

Platform devices and services provide device or service specific functionality in support of applications by way of the following components:

**Base Device Components:** DeviceComponent, LoadableDeviceComponent, ExecutableDeviceComponent and AggregateDeviceComponent, which provide management and control of hardware devices within the system;

**Framework Services Components:** PlatformComponentFactoryComponent, ServiceComponent or ManageableServiceComponent which provide additional support functions and services.

These components realize interfaces that are described below:

**Base Device Interfaces:** CapacityManagement, DeviceAttributes, AdministratableInterface, LoadableInterface, ExecutableInterface, AggregateDevice, and AggregateDeviceAttributes;

**Common Interfaces:** ComponentFactory;
Base Application Interfaces: ComponentIdentifier, ControllableInterface, PortAccessor, LifeCycle, TestableInterface and PropertySet.

2.2.4 Core Framework Control Architecture

The Core Framework is the essential set of open application-layer interfaces and services which provide an abstraction of the underlying system software and hardware. Core Framework Control provides management of domains consisting of the following SCA defined components:

Framework Control Components: ApplicationManagerComponent, ApplicationFactoryComponent, DomainManagerComponent, and DeviceManagerComponent, which control the instantiation, management, and destruction/removal of software from the system;

Framework Services Components: FileComponent, FileSystemComponent and FileManagerComponent, which provide additional support functions and services;

Common Components: BaseFactoryComponent.

These components realize interfaces described below:

Framework Control Interfaces: ApplicationManager, DeploymentAttributes, ApplicationFactory, ComponentRegistry, DomainInstallation, DomainManager, EventChannelRegistry, FullComponentRegistry, and ReleasableManager;

Framework Services Interfaces: File, FileSystem and FileManager;

Base Application Interfaces: ComponentIdentifier, ControllableInterface, PortAccessor, LifeCycle, TestableInterface and PropertySet;

Common Interfaces: ComponentFactory.

2.2.5 Structure

All SCA compliant systems require certain software components to be present in order to provide for component deployment, management, and interconnection. These include a DomainManagerComponent (including support for the ApplicationFactoryComponent and ApplicationManagerComponent), DeviceManagerComponent, FileManagerComponent, and FileSystemComponent. The management hierarchy of these entities is depicted in Figure 2-6.

An SCA compliant system includes a DomainManagerComponent which contains knowledge of all existing implementations installed or loaded onto the system including references to all file systems (through the file manager), device managers, application factories and applications.

Each DeviceManagerComponent, in turn, contains complete knowledge of a set of DeviceComponents and/or ServiceComponents. A system may have multiple DeviceManagerComponents but each DeviceManagerComponent registers with the DomainManagerComponent to assure that the DomainManagerComponent has complete cognizance of the system. A DeviceManagerComponent may have an associated FileSystemComponent (or FileManagerComponent to support multiple file systems) as indicated in the Figure 2-6.

An ApplicationManagerComponent, created by the ApplicationFactoryComponent, provides access to a specific application that is instantiated on the system.
2.2.6 Domain Profile

The SCA defines a set of files referred to as the Domain Profile, depicted in Figure 2-7, which describes the characteristics and attributes of the services, devices, and applications installed on the system. The Domain Profile is a hierarchical collection of descriptor files that define the properties of all software components in the system.

Each software element in the system is described by a Software Package Descriptor (SPD) and a Software Component Descriptor (SCD) file. An SPD file contains the details of a software module that are to be loaded and executed. The SPD provides identification of the software (title, author, etc.) as well as the name of the code file (executable, library or driver), implementation details (language, OS, etc.), dependencies to other SPDs and devices, and references to Properties Descriptor (PRF) and SCD files. The SCD defines interfaces supported and used by a specific component.

The Software Assembly Descriptor (SAD) file describes the composition and configuration of an application. The SAD references all SPDs and SADs needed for this application, defines required connections between application components (connection of provides and uses ports / interfaces), defines needed connections to devices and services, provides additional information on how to locate the needed devices and services, defines any co-location (deployment) dependencies, and identifies component(s) within the application as the application controller.
SAD may also reference an Application Deployment Descriptor (ADD) that defines the channel deployment precedence order for the application.

The Device Configuration Descriptor (DCD) identifies all devices and services associated with a device manager, by referencing its associated SPDs. The DCD also defines properties of the specific device manager, enumerates the needed connections to services (e.g. file systems), and provides additional information on how to locate the domain manager. A DCD may also contain a reference to a Device Package Descriptor (DPD) file which provides a detailed description of the associated hardware device.

The Domain Manager Configuration Descriptor (DMD) provides the location of the SPD file for a specific domain manager. The DMD also specifies connections to other software components (e.g. services) which are required by the domain manager. The DMD may also reference a Platform Deployment Descriptor (PDD) that describes the channels for a platform.

The PRF contains information about the properties applicable to a software package or a device package. A PRF provides information about the properties of a component such as its default values or configuration types.
3 SCA PLATFORM INDEPENDENT MODEL (PIM)

This section documents a platform independent representation of the SCA. Technology specific mappings of the SCA PIM are documented in Appendix E. OMG IDL is the standard representation for the standalone interface definitions within the SCA platform independent model.

3.1 OPERATING ENVIRONMENT

This section contains the requirements of the operating system, transfer mechanism, and the CF interfaces and operations that comprise the SCA Operating Environment.
3.1.1 Operating System

The processing environment and the functions performed in the architecture impose differing constraints on the architecture. Appendix B is defined to support portability of waveforms, scalability of the architecture, and commercial viability. POSIX specifications are used as a basis for this profile. The notional relationship of the OE and applications to Appendix B is depicted in Figure 3-1. SCA451 The OE shall provide the functions and options designated as mandatory by a profile defined in Appendix B. ^1 The OE is not limited to providing the functions and options designated as mandatory by the profile. OE implementations are not limited to using the services designated as mandatory by Appendix B.

![Figure 3-1: Notional Relationship of OE and Application to an SCA AEP](image)

SCA1 The OE and related file systems shall support a maximum filename length of 40 characters and a maximum pathname length of 1024 characters. Applications are limited to using the OS services that are designated as mandatory for the profile. Applications perform file access through the CF (application requirements are covered in section 3.1.3.2.2.1).

3.1.2 Transfer Mechanism & Services

SCA452 The OE shall provide a transfer mechanism that, at a minimum, provides the features specified in Appendix E for the specific platform technology implemented.

3.1.2.1 Log Service

An SCA compliant implementation may include a log service. SCA453 The log service shall conform to the OMG Lightweight Log Service Specification [1].

^1 The Appendix B version corresponds to the application component's SCA version
3.1.2.2 Event Service and Standard Events

3.1.2.2.1 Event Service

SCA454 The OE shall provide an event capability which implements the PushConsumer and PushSupplier interfaces of the CosEventComm module as described in OMG Event Service Specification [2] consistent with the IDL found in that specification.

The Event Service has the capability to create event channels. An event channel allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a producer of events. For example, event channels may be standard objects and communication through those channels is accomplished using standard requests. SCA3 The OE shall provide two standard event channels: Incoming Domain Management and Outgoing Domain Management. The Incoming Domain Management Channel name is "IDM_Channel". The Outgoing Domain Management Channel name is "ODM_Channel". The Incoming Domain Management event channel is used by components within the domain to generate events (e.g., device state change event) that are consumed by domain management components (e.g., ApplicationFactoryComponent, ApplicationManagerComponent, DomainManagerComponent, etc.). The Outgoing Domain Management Channel is used by domain clients (e.g., User Interface (UI)) to receive events (e.g., additions or removals from the domain) generated from domain management components. Besides these two standard event channels, the OE allows other event channels to be set up by application developers.

3.1.2.2.2 StandardEvent Module

The StandardEvent module specifies type definitions that are used for passing events from event producers to event consumers. The IDL for this module is specified in Appendix C of this specification.

3.1.2.2.3 Types

3.1.2.2.3.1 StateChangeCategoryType

The type StateChangeCategoryType is an enumeration that is utilized in the StateChangeEvent. It is used to identify the category of state change that has occurred.

```c
enum StateChangeCategoryType
{
    ADMINISTRATIVE_STATE_EVENT,
    OPERATIONAL_STATE_EVENT,
    USAGE_STATE_EVENT
};
```

3.1.2.2.3.2 StateChangeType

The type StateChangeType is an enumeration that is utilized in the StateChangeEvent. It is used to identify the specific states of the event source before and after the state change occurred.

```c
enum StateChangeType
{
    LOCKED,    /*Administrative State Event */
    UNLOCKED,  /*Administrative State Event */
    SHUTTING_DOWN, /*Administrative State Event */
};
```
ENABLED,  /*Operational State Event */
DISABLED, /*Operational State Event */
IDLE,     /*Usage State Event */
ACTIVE,   /*Usage State Event */
BUSY      /*Usage State Event */
};

3.1.2.2.3.3 StateChangeEventType
The type StateChangeEventType is a structure used to indicate that the state of the event source has changed.

```c
struct StateChangeEventType
{
    string             producerId;
    string             sourceId;
    StateChangeCategoryType stateChangeCategory;
    StateChangeType    stateChangeFrom;
    StateChangeType    stateChangeTo;
};
```

3.1.2.2.3.4 ComponentChangeType
The type ComponentChangeType is an enumeration that is utilized in the ComponentChangeEvent. It is used to indicate an object that has been added to or removed from the domain.

```c
enum ComponentChangeType
{
    ADDED,
    REMOVED
};
```

3.1.2.2.3.5 ComponentChangeEvent
The type ComponentChangeEvent is a structure used to indicate that an event source has been added to or removed from the domain.

```c
struct ComponentChangeEvent
{
    string             producerId;
    ComponentChangeType componentChange;
    CF::ComponentType  domainComponent
};
```

When an object is removed, the CF::ComponentType's componentObject field should be a nil reference and its providesPorts field should be a zero length sequence.

3.1.2.3 Additional Services
The OE may include services other than those (i.e. log, file system, and event services) defined within the SCA. Those additional services may be deployed by a DeviceManagerComponent and managed by the framework through the CF based interfaces.
Service definitions should consist of APIs, behavior, state, priority and additional information in order to establish a clear contract between the service provider and user. IDL is the technology used to represent the service interfaces in order to foster reuse, extensibility and interoperability among SCA components.

3.1.3 Core Framework

This section includes a detailed description of the purpose of each CF interface, component, the purpose of each supported operation within the interface, and interface class diagrams to support these descriptions. The corresponding IDL for the CF is specified in Appendix C.

Figure 3-2 depicts the key elements of the CF and the UML relationships between those elements. A DomainManagerComponent manages the software applications, application factories, hardware devices (represented by software devices) and device managers within the system. Some software components may directly control the system's internal hardware devices; these components are logical devices, which implement the Base Device interfaces. Other software components have no direct relationship with a hardware device, but perform application services for the user and may implement the Base Application interfaces. These interfaces provide a consistent way of configuring and tearing down these components. Each component can potentially communicate with other components. An application is a collection of one or more ApplicationComponents which provide a specific service or function that is managed through the ApplicationManager interface. The resources of an application are allocated to one or more hardware devices by the application factory based upon various factors including the current availability of hardware devices, the component's behavior rules, and the loading requirements of each component. The components may then be created via the ComponentFactory interface or through a DeviceComponent and connected to other system components.
The file service interfaces (FileManager, FileSystem, and File) are used for installation and removal of application files, and for loading and unloading application files on the various processors that the devices execute upon.

### 3.1.3.1 Common Elements

#### 3.1.3.1.1 Interfaces

The Common Interfaces provide abstractions for common features, constraints and associations of SCA products that will be utilized by device, service or application developers.

3.1.3.1.1.1 ComponentFactory

3.1.3.1.1.1 Description

The ComponentFactory interface provides an optional mechanism for the management (i.e. creation and tear down) of components. The ComponentFactory interface is designed after the Factory Design Patterns [10]. The ComponentFactory interface UML is depicted in Figure 3-3.
3.1.3.1.1.1.2 UML

![ComponentFactory Interface UML](image)

**Figure 3-3: ComponentFactory Interface UML**

3.1.3.1.1.3 Types
3.1.3.1.1.3.1 CreateComponentFailure
The CreateComponentFailure exception indicates that the `createComponent` operation failed to create the component. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```java
class CreateComponentFailure { ErrorNumberType errorNumber;
string msg; }
```

3.1.3.1.1.4 Attributes
N/A.

3.1.3.1.1.5 Operations
3.1.3.1.1.5.1 createComponent
3.1.3.1.1.5.1.1 Brief Rationale
The `createComponent` operation provides the capability to create components in the same process space as the component factory. This behavior is an alternative approach to the `ExecutableInterface::execute` operation for creating a component.

3.1.3.1.1.5.1.2 Synopsis

```java
ComponentType createComponent (in string componentId, in Properties qualifiers) raises (CreateComponentFailure);
```

3.1.3.1.1.5.1.3 Behavior
The componentId parameter is the identifier for a component. The qualifiers parameter contains values used by the component factory in creation of the component. The qualifiers may be used to identify, for example, specific subtypes of components created by a component factory.

SCA386 The `createComponent` operation shall create a component if no component exists for the given componentId.  
SCA387 The `createComponent` operation shall assign the given componentId to a new component.
3.1.3.1.1.5.1.4 Returns

SCA388 The `createComponent` operation shall return a CF::ComponentType structure. The contents of the returned CF::ComponentType's identifier, componentObject, profile, and type fields are as specified in section 3.1.3.1.3.17, CF::ComponentType. The returned CF::ComponentType does not have to populate its providesPorts field.

3.1.3.1.1.5.1.5 Exceptions/Errors

SCA389 The `createComponent` operation shall raise the CreateComponentFailure exception when it cannot create the component or the component already exists.

3.1.3.1.2 Components

The Common Components provide abstractions for common features, constraints and associations of SCA products that will be utilized by device, service or application developers.

3.1.3.1.2.1 BaseComponent

3.1.3.1.2.1.1 Description

A BaseComponent is an abstract component that extends the UML component. BaseComponent provides an abstraction for the core associations and requirements that are used by many of the SCA components.

![Component Properties Diagram](image)

**Figure 3-4:** BaseComponent UML
3.1.3.1.2.1.2 Associations

- domainProfile: A BaseComponent is associated with a SPD and zero to many other domain profile files.
- eventChannel: A BaseComponent produces and consumes event messages to and from event channels.
- targetLog: A BaseComponent produces log messages and sends them to system log(s).
- property: A BaseComponent's configuration is dictated via the categories of configure properties contained within its domainProfile.
- connectedComponent: A BaseComponent can be connected with and leverage capabilities provided by other BaseComponent(s).
- registrar: A BaseComponent registers its reference via the ComponentRegistry interface.

3.1.3.1.2.1.3 Semantics

SCA420 A BaseComponent shall implement a 'configure' kind of property with a name of PRODUCER_LOG_LEVEL. The PRODUCER_LOG_LEVEL configure property provides the ability to filter the log message output of a component. This property may be configured via the PropertySet interface to output only specific log levels. SCA421 A BaseComponent shall output only those log records to a log service that correspond to enabled log level values in the PRODUCER_LOG_LEVEL attribute. Log levels that are not in the PRODUCER_LOG_LEVEL attribute are disabled. A BaseComponent uses its identifier in the producerId field of the log record output to the log service. SCA423 A BaseComponent shall operate normally in the case where the connections to a log service are nil or an invalid reference.

The PropertySet configure and query, TestableInterface::runTest, and ControllableInterface::start operations are not inhibited by the ControllableInterface::stop operation. SCA518 The releaseObject operation shall disconnect any ports that are still connected.

The CosEventComm module is used by consumers for receiving events and by producers for generating events. SCA444 A BaseComponent (e.g., ManageableApplicationComponent, DomainManagerComponent, etc.) that consumes events shall implement the CosEventComm::PushConsumer interface. SCA424 A BaseComponent that produces events shall implement the CosEventComm::PushSupplier interface and use the CosEventComm::PushConsumer interface for generating the events. SCA425 A producer BaseComponent shall not forward or raise any exceptions when the connection to a CosEventComm::PushConsumer is a nil or invalid reference.

A BaseComponent may additionally implement and use component-specific interfaces for data and/or control. Interfaces provided by a BaseComponent are described in an SCD file as provides ports. Interfaces required by a BaseComponent are described in a SCD file as uses ports. A BaseComponent may register its component reference via the ComponentRegistry interface. The registering BaseComponent sets its CF::ComponentType's identifier, componentObject, and type fields as specified in section 3.1.3.1.3.17, CF::ComponentType. A registering BaseComponent does not have to set its CF::ComponentType's providesPorts field.
3.1.3.1.2.1.4 Constraints
SCA426 A BaseComponent shall realize the ComponentIdentifier interface.
SCA427 A BaseComponent shall be associated with a domain profile file.
SCA428 A BaseComponent shall provide a test implementation for all properties whose kindtype is "test" as defined in its descriptor files.
SCA429 A BaseComponent shall configure or retrieve query values for all properties whose kindtype is "configure" as defined in its domain profile. Configure properties are configure readwrite and writeonly properties. Query properties are all configure properties whose mode element is "readwrite" or "readonly" and any allocation properties with an action value of "external".
SCA430 A BaseComponent shall supply ports for all the ports defined in its domain profile.
SCA432 A BaseComponent shall realize the LifeCycle interface. The LifeCycle operations are used during deployment and teardown of a component. An initialized BaseComponent may not be releasable.
SCA433 A BaseComponent shall realize the ControllableInterface interface to provide overall management control of the component.
SCA545 A BaseComponent shall realize the PropertySet interface to configure and query its properties.
SCA546 A BaseComponent shall realize the TestableInterface interface to define and utilize its test properties.
SCA547 A BaseComponent shall realize the PortAccessor interface as a proxy for its uses and provides ports.
SCA548 A BaseComponent shall implement its optional composition relationships via inheritance.
3.1.3.1.2.2 BaseFactoryComponent
3.1.3.1.2.2.1 Description
A BaseFactoryComponent is an abstract component which provides an optional mechanism that may be used to create components. The factory mechanism provides client-server isolation among components and provides a standard mechanism of obtaining a component without knowing its identity.
3.1.3.1.2.2.2 Associations

- createdComponent: A BaseFactoryComponent provides a mechanism to create new component instances.

3.1.3.1.2.2.3 Semantics

A BaseFactoryComponent is used to create a component. The BaseFactoryComponent provides the mechanism that may be used to create separate process threads for each component instantiated by the factory. A BaseFactoryComponent should contain a collection of configurable initialization and component creation properties.

SCA540 Each BaseFactoryComponent shall support the mandatory Component Identifier execute parameter as described in section 3.1.3.1.3.5.1, in addition to their user-defined execute properties in the component's SPD. SCA541 Each executable BaseFactoryComponent shall set its identifier attribute using the Component Identifier execute parameter.

SCA574 The releaseObject operation shall release all component instances created by the BaseFactoryComponent.

3.1.3.1.2.2.4 Constraints

SCA413 A BaseFactoryComponent shall realize the ComponentFactory interface. SCA414 A BaseFactoryComponent shall fulfill the BaseComponent requirements. SCA549 A BaseFactoryComponent shall realize the LifeCycle interface.

3.1.3.1.3 Core Framework Base Types

The CF Base Types are the underlying types used in the CF interfaces.

3.1.3.1.3.1 DataType

This type defines a structure, which may be used to specify an id and value pair. The id field is the identifier of the element being defined. The value field contains the value which is associated with the defined id.

```cpp
struct DataType {
    string id;
    any value;
};
```

3.1.3.1.3.2 ObjectSequence

The CF::ObjectSequence type defines an unbounded sequence of objects.

```cpp
typedef sequence <CF::OctetSeq> ObjectSequence;
```

3.1.3.1.3.3 FileException

The CF::FileException exception indicates a file-related error occurred. The error number indicates a CF::ErrorNumberType value. The message provides information describing the error. The message may be used for logging the error.

```cpp
exception FileException {ErrorNumberType errorNumber; string msg; }
```
3.1.3.1.3.4 InvalidFileName
The CF::InvalidFileName exception indicates an invalid file name was passed to a file service operation. The error number indicates a CF::ErrorNumberType value. The message provides information describing why the file name was invalid.

exception InvalidFileName {ErrorNumberType errorNumber; string msg; }

3.1.3.1.3.5 InvalidObjectReference
The CF::InvalidObjectReference exception indicates an invalid object reference error.

exception InvalidObjectReference {string msg; }

3.1.3.1.3.6 InvalidProfile
The CF::InvalidProfile exception indicates an invalid profile error.

exception InvalidProfile{}

3.1.3.1.3.7 OctetSequence
This type is an unbounded sequence of octets.

typedef sequence <CF::OctetType> OctetSequence;

3.1.3.1.3.8 Properties
The CF::Properties is an IDL unbounded sequence of CF::DataType(s), which is used in defining a sequence of name and value pairs.

typedef sequence <DataType> Properties;

3.1.3.1.3.9 StringSequence
This type defines a sequence of strings.

typedef sequence <string> StringSequence;

3.1.3.1.3.10 UnknownProperties
The CF::UnknownProperties exception indicates the unsuccessful retrieval of a component's properties. The invalidProperties returned indicate the properties that were unknown.

exception UnknownProperties {Properties invalidProperties; }

3.1.3.1.3.11 DeviceAssignmentType
The CF::DeviceAssignmentType defines a structure that associates a component with the device which the component either uses, is loaded upon or on which it is executed.

struct DeviceAssignmentType
{
    string componentId;
    string assignedDeviceId;
};

3.1.3.1.3.12 DeviceAssignmentSequence
The IDL sequence, CF::DeviceAssignmentSequence, provides an unbounded sequence of CF::DeviceAssignmentTypes.

typedef sequence <DeviceAssignmentType>DeviceAssignmentSequence;
3.1.3.1.3.13 ErrorNumberType.
This enum is used to pass error number information in various exceptions. Those exceptions starting with "CF_E" map the POSIX definitions (with the "CF_" removed), and is found in reference [9]. CF_NOTSET is not defined in the POSIX specification. CF_NOTSET is an SCA specific value that is applicable for any exception when the method specific or standard POSIX error values are not appropriate.

```c
enum ErrorNumberType
{
    CF_NOTSET, CF_E2BIG, CF_EACCES, CF_EAGAIN, CF_EBADF, CF_EBADMSG,
    CF_EBUSY, CF_ECANCELED, CF_ECHILD, CF_EDEADLK, CF_EDOM,
    CF_EEXIST, CF_EFAULT, CF_FBIG, CF_FPROGRESS,
    CF_EINVAL, CF_EIO, CF_EISDIR, CF_EMFILE, CF_EMLNK,
    CF_EMFSIZE, CF_ENAMETOOLONG, CF_ENFILE, CF_ENODEV, CF_ENOENT,
    CF_ENOEXEC, CF_ENOLCK, CF_ENOMEM, CF_ENOSPC, CF_ENOSYS,
    CF_ENOTDIR, CF_ENOTEMPTY, CF_ENOTSUP, CF_ENOTTY, CF_ENXIO,
    CF_EPERM, CF_ESPIPE, CF_ESRCH, CF_ESRCH, CF_ETERMOUT, CF_EXDEV
};
```

3.1.3.1.3.14 PortAccessType
The CF::PortAccessType structure defines a port. The portName field is the name of the port. The portReference field is object reference of the port.

```c
struct PortAccessType
{
    string portName;
    Object portReference;
};
```

3.1.3.1.3.15 Ports
The CF::Ports type defines a name/value sequence of CF::PortAccessType structures.

```c
typedef sequence <PortAccessType> Ports;
```

3.1.3.1.3.16 ComponentEnumType
The CF::ComponentEnumType enumeration defines the basic type of a component. The APPLICATION_COMPONENT identifier is a component which is deployed as part of a Software Assembly. The MANAGEABLE_APPLICATION_COMPONENT identifier is an ApplicationComponent, deployed as part of a Software Assembly, that the framework can manage through the CF based interfaces. The DEVICE_COMPONENT identifier represents a DeviceComponent deployed by a DeviceManagerComponent. The LOADABLE_DEVICE_COMPONENT identifier represents a LoadableDeviceComponent deployed by a DeviceManagerComponent. The EXECUTABLE_DEVICE_COMPONENT identifier represents an ExecutableDeviceComponent deployed by a DeviceManagerComponent. The MANAGEABLE_SERVICE_COMPONENT identifier is a ServiceComponent, deployed by a DeviceManagerComponent, that the framework can manage through the CF based interfaces. The SERVICE_COMPONENT identifier is a ServiceComponent, deployed by a
DeviceManagerComponent, that could possibly implement any interface (e.g. Log, FileSystem, etc.). The DEVICE_MANAGER_COMPONENT identifier represents a DeviceManagerComponent which deploys devices and services. The DOMAIN_MANAGER_COMPONENT identifier represents a DomainManagerComponent which manages all of the components within a domain. The APPLICATION_MANAGER_COMPONENT identifier represents an ApplicationManagerComponent which provides the proxy to enable communication between the Core framework and an application. The APPLICATION_FACTORY_COMPONENT identifier represents an ApplicationFactoryComponent which creates applications. The APPLICATION_COMPONENT_FACTORY_COMPONENT identifier represents an ApplicationComponentFactoryComponent which creates application components. The PLATFORM_COMPONENT_FACTORY_COMPONENT identifier represents a PlatformComponentFactoryComponent which creates device or service components.

```c
enum ComponentEnumType
{
    APPLICATION_COMPONENT,
    MANAGEABLE_APPLICATION_COMPONENT,
    DEVICE_COMPONENT,
    LOADABLE_DEVICE_COMPONENT,
    EXECUTABLE_DEVICE_COMPONENT,
    MANAGEABLE_SERVICE_COMPONENT,
    SERVICE_COMPONENT,
    DEVICE_MANAGER_COMPONENT,
    DOMAIN_MANAGER_COMPONENT,
    APPLICATION_MANAGER_COMPONENT,
    APPLICATION_FACTORY_COMPONENT,
    APPLICATION_COMPONENT_FACTORY_COMPONENT,
    PLATFORM_COMPONENT_FACTORY_COMPONENT
};
```

3.1.3.1.3.17 ComponentType

The CF::ComponentType structure defines the basic elements of a component.

The identifier field is the id of the component as specified through its createComponent/execute parameters.

The profile field is either the component's profile filename or the profile itself (the DomainManagerComponent uses a DMD, the ApplicationFactoryComponent and ApplicationManagerComponent use a SAD, the DeviceManagerComponent uses a DCD and all of the other components use an SPD). The application's SAD filename is an absolute pathname relative to a mounted FileSystemComponent and the file is obtained via the DomainManagerComponent's FileManagerComponent. ApplicationComponent Files referenced within the profile are obtained via a DomainManagerComponent's FileManagerComponent. The DCD and BasePlatformComponent filename is an absolute pathname relative to a mounted FileSystemComponent and the file is obtained via a DeviceManagerComponent's FileSystemComponent.
The type field is the value of the SCD componentType element described in Appendix D-1 that maps to CF::ComponentEnumType.

The componentObject field is the object reference of the component.

The providesPorts field is a sequence containing the ports provided by the component.

The SpecializedInfo field is a variant and is used by to provide supplemental, component specific information.

```c
struct ComponentType {
    string identifier;
    string profile;
    CF::ComponentEnumType type;
    Object componentObject;
    CF::Ports providesPorts;
    CF::Properties specializedInfo;
};
```

3.1.3.1.3.18 Components

The CF::Components type defines a sequence of CF::ComponentType structures.

```c
typedef sequence <ComponentType> Components;
```

3.1.3.1.3.19 InvalidState

The CF::InvalidState exception indicates that the device is not capable of executing the requested behavior due to the device's current state. The message is component-dependent, providing additional information describing the reason for the error.

```c
exception InvalidState {string msg;};
```

3.1.3.1.3.20 PropertyActionType

The CF::PropertyActionType enumeration defines the actions associated with an allocation property. The property action types are equivalent to the values of the action element described in Appendix D-1.

```c
enum PropertyActionType {
    CF_EQ, CF_NE, CF_GT, CF_GE, CF_LT, CF_LE, CF_EXTERNAL
};
```

3.1.3.1.3.21 PropertyType

The CF::PropertyType enumeration defines the representations that are used to describe allocation property data types. The property types are equivalent to those the type attribute of the simple or simplesequence elements described in Appendix D-1.

```c
enum PropertyType {
    CF_BOOLEAN, CF_CHAR, CF_DOUBLE, CF_FLOAT, CF_SHORT, CF_LONG,
    CF_OBJREF, CF_OCTET, CF_STRING, CF_USHORT, CF_ULONG
};
```
3.1.3.1.3.22 AllocationPropertyType

The CF::AllocationPropertyType structure defines the basic information to store a *simple* or *sequence* element allocation property. The id field is the property's id attribute. The values field is the sequence of property value(s). The action field is the type of action associated with the property. The type field is the data type of the property.

```cpp
class AllocationPropertyType {
    string id;
    CF::StringSequence values;
    CF::PropertyActionType action;
    CF::PropertyType type;
}
```

3.1.3.1.3.23 AllocationProperties

The CF::AllocationProperties sequence defines a sequence of CF::AllocationPropertyType structures which provides supplemental information for a BasePlatformComponent's specializedInfo. The associated ID for CF::AllocationProperties type is ALLOCATION_PROPS_ID.

```cpp
typedef sequence< AllocationPropertyType>  AllocationProperties;
```

3.1.3.1.3.24 ManagerInfo

The CF::ManagerInfo structure defines the supplemental information that is used as the CF::ComponentType's specializedInfo of a manager component. The fileSys field is the file system used by this manager component. The deployedComponents field is a sequence of components that have been deployed by this manager component. The associated ID for the CF::ManagerInfo type is MANAGER_INFO_ID. For each deployed component, it's CF::ComponentType's identifier, componentObject, profile, providesPorts, and type fields are as specified in section 3.1.3.1.3.17, CF::ComponentType.

For each deployed BaseComponent in deployedComponents, the deployed component specializedInfo contains deployment information consisting of the following:

1. A CF::DataType with an id of EXECUTION_ID and value of ExecutableInterface::ExecutionID_Type;
2. A CF::DataType with an id of IMPLEMENTATION_ID and value of the profile's implementation id used for deployment;
3. A CF::DataType with an id of TARGET_DEVICE_ID and value of the DeviceComponent's identifier that component is deployed on;
4. A CF::DataType with an id of USES_DEVICE_ID and a value of CF::UsesDeviceAssignmentSequence type if a component has uses devices as indicated by its profile.

```cpp
class ManagerInfo {
    CF::FileSystem fileSys;
    CF::Components deployedComponents;
}
```
3.1.3.1.3.25 SpecializedInfo Identifiers
The constants defined in this section identify the type value for CF::DataTypes used within a CF::ComponentType's specializedInfo field.
The DEVICE_MANAGER_ID constant is used as the id field of a DeviceManagerComponent within a BasePlatformComponent.
const string DEVICE_MANAGER_ID = "DEVICE_MANAGER_ID";
The MANAGER_INFO_ID constant is used as the id field of a CF::ManagerInfo type.
const string MANAGER_INFO_ID = "MANAGER_INFO";
The EXECUTION_ID constant is used as the id field of an ExecutionID_Type.
const string EXECUTION_ID = "EXECUTION_ID";
The ALLOCATION_PROPS_ID constant is used as the id field of a CF::AllocationProperties type.
const string ALLOCATION_PROPS_ID = "ALLOCATION_PROPS";
The IMPLEMENTATION_ID constant is used as the id field of an SPD implementation id. The identifier references the implementation used for creating of the component.
const string IMPLEMENTATION_ID = "IMPLEMENTATION_ID";
The TARGET_DEVICE_ID constant is used as the id field of the device identifier. This references the device that deployed the component.
const string TARGET_DEVICE_ID = "TARGET_DEVICE";
The USES_DEVICE_ID constant is used as the id field of the CF::UsesDeviceAssignmentSequence. This sequence denotes the devices used by the component.
const string USES_DEVICE_ID = "USESDEVICE";
The COMPONENTS_ID constant is used as the id field of the CF::Components type.
const string COMPONENTS_ID = "COMPONENTS";

3.1.3.1.3.26 UsesDeviceAssignmentType
This structure associates a component's profile uses device identifier with the assigned device identifier.
struct UsesDeviceAssignmentType
{
    string usesDeviceId;
    string assignedDeviceId;
};

3.1.3.1.3.27 UsesDeviceAssignmentSeq
The CF::UsesDeviceAssignmentSeq sequence specifies an unbounded sequence of 0..n UseDeviceAssignmentType structures. The CF::UsesDeviceAssignmentSeq contains supplemental deployment information for a CF::ComponentType's specializedInfo. The associated ID for UsesDeviceAssignmentSeq type is USES_DEVICE_ID.
typedef sequence <UsesDeviceAssignmentType> UsesDeviceAssignmentSeq;

3.1.3.2 Base Application

3.1.3.2.1 Interfaces

Base Application Interfaces are defined by the Core Framework requirements and implemented by application developers; see section 3.1.3.2.2.1 for application requirements. Base Application Interfaces are implemented using the appropriate Platform Specific interface definitions presented in Appendix E.

3.1.3.2.1.1 ComponentIdentifier

3.1.3.2.1.1.1 Description

The ComponentIdentifier interface provides a readonly identifier attribute for a component. The interface for a ComponentIdentifier is based upon its identifier attribute, which is the instance-unique identifier for this component.

3.1.3.2.1.1.2 UML

![ComponentIdentifier Interface UML](image)

Figure 3-6: ComponentIdentifier Interface UML

3.1.3.2.1.1.3 Types

N/A.

3.1.3.2.1.1.4 Attributes

3.1.3.2.1.1.4.1 identifier

SCA6 The readonly identifier attribute shall return the instance-unique identifier for a component.

readonly attribute string identifier;

3.1.3.2.1.1.5 Operations

N/A.

3.1.3.2.1.2 PortAccessor

3.1.3.2.1.2.1 Description

This interface provides operations for managing associations between ports. The PortAccessor interface UML is depicted in Figure 3-7. An application establishes the operations for transferring data and control. The application also establishes the meaning of the data and control values. Examples of how applications may use ports in different ways include: push or pull, synchronous or asynchronous, mono- or bi-directional, or whether to use flow control (e.g., pause, start, and stop).
The nature of a PortAccessor instance's fan-in, fan-out, or one-to-one relationships is component dependent. How a components' ports are connected is described in the SAD and the DCD files of the Domain Profile (3.1.3.6).

3.1.3.2.1.2.2 UML

![UML Diagram]

**Figure 3-7: PortAccessor Interface UML**

3.1.3.2.1.2.3 Types

3.1.3.2.1.2.3.1 ConnectionType

The ConnectionType structure defines a type for information needed to make a connection. The connectionId field is the id of the connection. The portReference field is the object reference of the provided port.

```c
struct ConnectionType {
    ConnectionIdType portConnectionId;
    Object portReference;
};
```

3.1.3.2.1.2.3.2 Connections

The Connections type defines a sequence of ConnectionType structures.

```c
typedef sequence <ConnectionType> Connections;
```

3.1.3.2.1.2.3.3 ConnectionErrorType

The ConnectionErrorType structure identifies a port and associated error code to be provided in the InvalidPort exception defined in 3.1.3.2.1.2.3.6.

```c
struct ConnectionErrorType {
    ConnectionIdType portConnectionId;
    unsigned short errorCode;
};
```

3.1.3.2.1.2.3.4 ConnectionIdType

The ConnectionIdType structure defines a type for information needed to describe a connection. The connectionId field is the id of the connection. The portName field is the name of the (uses or provides) port.

```c
struct ConnectionIdType {
    string connectionId;
};
```
string portName;
};

3.1.3.2.1.2.3.5 Disconnections
The Disconnections type defines a sequence of ConnectionIdType structures.
typedef sequence <ConnectionIdType> Disconnections;

3.1.3.2.1.2.3.6 InvalidPort
The InvalidPort exception indicates one of the following errors has occurred in the specification of a connection:
1. An errorCode of 1 indicates the provides portReference is invalid (e.g. unable to narrow object reference) or illegal object reference,
2. An errorCode of 2 indicates the connectionId is invalid,
3. An errorCode of 3 indicates uses or provides port portName does not exist for the given connectionId, or
4. An errorCode of 4 indicates the port has reached its maximum number of connections and is unable to accept any additional connections.

exception InvalidPort {ConnectionErrorType invalidConnections};

3.1.3.2.1.2.4 Attributes
N/A.

3.1.3.2.1.2.5 Operations
3.1.3.2.1.2.5.1 connectUsesPorts
3.1.3.2.1.2.5.1.1 Brief Rationale
Applications require the connectUsesPorts operation to establish associations between ports. Ports provide channels through which data and/or control pass.
The connectUsesPorts supplies a component with a sequence of connection information. The input portConnections parameter is a sequence of connectionIds, uses port names, and provides port object references.

3.1.3.2.1.2.5.1.2 Synopsis
void connectUsesPorts (in Connections portConnections) raises (InvalidPort);

3.1.3.2.1.2.5.1.3 Behavior
SCA7 The connectUsesPorts operation shall make the connection(s) to the component identified by its input portConnections parameter. A port may support several connections. The resulting portConnectionIdS are used by the disconnectPorts operation when breaking specific connection(s). SCA519 The connectUsesPorts operation shall disconnect any connections it formed if any connections in the input portConnections parameter cannot be successfully established.

3.1.3.2.1.2.5.1.4 Returns
This operation does not return a value.

3.1.3.2.1.2.5.1.5 Exceptions/Errors
SCA8 The `connectUsesPorts` operation shall raise the `InvalidPort` exception when the input `portConnections` parameter provides an invalid connection for the specified port.

3.1.3.2.1.2.5.2 `disconnectPorts`

3.1.3.2.1.2.5.2.1 Brief Rationale

Applications require the `disconnectPorts` operation to allow consumer/producer components to disassociate themselves from their counterparts (consumer/producer).

The `disconnectPorts` operation releases a sequence of uses or provides ports from the connection(s). The input `portDisconnections` is a sequence of `connectionIds` and (uses or provides) port names.

3.1.3.2.1.2.5.2.2 Synopsis

```c
void disconnectPorts (in Disconnections portDisconnections) raises (InvalidPort);
```

3.1.3.2.1.2.5.2.3 Behavior

SCA10 The `disconnectPorts` operation shall break the connection(s) to the component identified by the input `portDisconnections` parameter.

SCA11 The `disconnectPorts` operation shall release all ports if the input `portDisconnections` parameter is a zero length sequence.

3.1.3.2.1.2.5.2.4 Returns

This operation does not return a value.

3.1.3.2.1.2.5.2.5 Exceptions/Errors

SCA12 The `disconnectPorts` operation shall raise the `InvalidPort` exception when the input `portDisconnections` parameter provides an unknown connection to the `PortAccessor`'s component.

3.1.3.2.1.2.5.3 `getProvidesPorts`

3.1.3.2.1.2.5.3.1 Brief Rationale

The `getProvidesPorts` operation provides a mechanism to obtain specific provides ports. The exact number of ports is specified in the component's software profile SCD (section 3.1.3.6). The input/output `portConnections` is a sequence of `connectionIds` and provides port object references.

3.1.3.2.1.2.5.3.2 Synopsis

```c
void getProvidesPorts (inout Connections portConnections) raises (InvalidPort);
```

3.1.3.2.1.2.5.3.3 Behavior

The `getProvidesPorts` operation returns the provides port object references associated with the component's input port names that are stated in its SCD.

3.1.3.2.1.2.5.3.4 Returns

SCA13 The `getProvidesPorts` operation shall return the object references that are associated with the input port names and the `connectionIds`.

3.1.3.2.1.2.5.3.5 Exceptions/Errors
SCA14 The `getProvidesPorts` operation shall raise an `InvalidPort` exception when the input `portConnections` parameter requests undefined connection(s).

3.1.3.2.1.3 LifeCycle

3.1.3.2.1.3.1 Description

The `LifeCycle` interface defines the generic operations for initializing or releasing instantiated component-specific data and/or processing elements. The `LifeCycle` interface UML is depicted in Figure 3-8.

### UML

```
«interface»
CF::LifeCycle
+ initialize(): void
+ releaseObject(): void
```

![Figure 3-8: LifeCycle Interface UML](image.png)

3.1.3.2.1.3.3 Types

3.1.3.2.1.3.3.1 InitializeError

The InitializeError exception indicates an error occurred during component initialization. `ErrorMessage` is component-dependent, providing additional information describing the reason why the error occurred.

```java
exception InitializeError { StringSequence errorMessage; }
```

3.1.3.2.1.3.3.2 ReleaseError

The ReleaseError exception indicates an error occurred during the component `releaseObject` operation. `ErrorMessage` is component-dependent, providing additional information describing the reason why the error occurred.

```java
exception ReleaseError { StringSequence errorMessage; }
```

3.1.3.2.1.3.4 Attributes

N/A.

3.1.3.2.1.3.5 Operations

3.1.3.2.1.3.5.1 initialize

3.1.3.2.1.3.5.1.1 Brief Rationale

The purpose of the `initialize` operation is to provide a mechanism to set a component to a known initial state. For example, data structures may be set to initial values, memory may be allocated, component may be configured to some state, etc.

3.1.3.2.1.3.5.1.2 Synopsis

```java
void initialize() raises (InitializeError);
```

3.1.3.2.1.3.5.1.3 Behavior
Initialization behavior is implementation dependent.

3.1.3.2.1.3.5.1.4 Returns
This operation does not return a value.

3.1.3.2.1.3.5.1.5 Exceptions/Errors
SCA15 The initialize operation shall raise an InitializeError exception when an initialization error occurs.

3.1.3.2.1.3.5.2 releaseObject
3.1.3.2.1.3.5.2.1 Brief Rationale
The purpose of the releaseObject operation is to provide a means by which an instantiated component may be torn down. There is client side and server side representation of instantiated component. The releaseObject operation provides the mechanism for releasing the instantiated component from the OE on the server side. The client has the responsibility to release its client side reference of the instantiated component.

3.1.3.2.1.3.5.2.2 Synopsis
void releaseObject() raises (ReleaseError);

3.1.3.2.1.3.5.2.3 Behavior
SCA16 The releaseObject operation shall release all internal memory allocated by the component during the life of the component.
SCA17 The releaseObject operation shall tear down the component and release it from the operating environment. Tearing down a component implies its port connections have been disconnected and all component ports and interfaces have been deactivated and terminated.

3.1.3.2.1.3.5.2.4 Returns
This operation does not return a value.

3.1.3.2.1.3.5.2.5 Exceptions/Errors
SCA18 The releaseObject operation shall raise a ReleaseError exception when a release error occurs.

3.1.3.2.1.4 TestableInterface
3.1.3.2.1.4.1 Description
The TestableInterface interface defines an operation that is used to test a component implementation. The TestableInterface interface UML is depicted in Figure 3-9.

3.1.3.2.1.4.2 UML

```
«interface»
CF::TestableInterface
+ runTest(unsigned long, Properties*): void
```

**Figure 3-9: TestableInterface Interface UML**
3.1.3.2.1.4.3 Types
3.1.3.2.1.4.3.1 UnknownTest
The UnknownTest exception indicates the input testId parameter is not known by the component.
exception UnknownTest{};
3.1.3.2.1.4.4 Attributes
N/A.
3.1.3.2.1.4.5 Operations
3.1.3.2.1.4.5.1 runTest
3.1.3.2.1.4.5.1.1 Brief Rationale
The runTest operation allows components to be blackbox tested. This allows built-in tests (BITs) to be implemented which provide a means to isolate faults (both software and hardware) within the system.
3.1.3.2.1.4.5.1.2 Synopsis
void runTest (in unsigned long testId, inout Properties testValues) raises (UnknownTest, UnknownProperties);
3.1.3.2.1.4.5.1.3 Behavior
SCA19 The runTest operation shall use the input testId parameter to determine which of its predefined test implementations should be performed. The id/value pair(s) of the testValues parameter should be used to provide additional information to the implementation-specific test to be run. SCA21 The runTest operation shall return the result(s) of the test in the testValues parameter.
Tests to be implemented by a component are component-dependent and are specified in the component's PRF. The testId parameter corresponds to the XML attribute testId of the property element test in a propertyfile.
The runTest operation does not execute any testing when the input testId or any of the input testValues are not known by the component or are out of range.
3.1.3.2.1.4.5.1.4 Returns
This operation does not return a value.
3.1.3.2.1.4.5.1.5 Exceptions/Errors
SCA23 The runTest operation shall raise the UnknownTest exception when there is no underlying test implementation that is associated with the input testId given.
SCA24 The runTest operation shall raise the CF::UnknownProperties exception when the input parameter testValues contains any CF::DataTypes that are not known by the component's test implementation or any values that are out of range for the requested test. SCA25 The exception parameter invalidProperties shall contain the invalid testValues properties id(s) that are not known by the component or the value(s) are out of range.
3.1.3.2.1.5 PropertySet

3.1.3.2.1.5.1 Description

The PropertySet interface defines configure and query operations to access component properties/attributes. The PropertySet interface UML is depicted in Figure 3-10.

3.1.3.2.1.5.2 UML

```
<interface>
CF::PropertySet
+ configure(Properties): void
+ query(Properties*): void
```

Figure 3-10: PropertySet Interface UML

3.1.3.2.1.5.3 Types

N/A.

3.1.3.2.1.5.3.1 InvalidConfiguration

The InvalidConfiguration exception indicates the configuration of a component has failed (no configuration at all was done). The message is component-dependent, providing additional information describing the reason why the error occurred. The invalidProperties returned indicate the properties that were invalid.

```c
exception InvalidConfiguration { string msg; Properties invalidProperties; }
```

3.1.3.2.1.5.3.2 PartialConfiguration

The PartialConfiguration exception indicates the configuration of a component was partially successful. The invalidProperties returned indicate the properties that were invalid.

```c
exception PartialConfiguration { Properties invalidProperties; }
```

3.1.3.2.1.5.4 Attributes

N/A.

3.1.3.2.1.5.5 Operations

3.1.3.2.1.5.5.1 configure

3.1.3.2.1.5.5.1.1 Brief Rationale

The configure operation allows id/value pair configuration properties to be assigned to components implementing this interface.

3.1.3.2.1.5.5.1.2 Synopsis

```c
void configure (in Properties configProperties) raises (InvalidConfiguration, PartialConfiguration);
```

3.1.3.2.1.5.5.1.3 Behavior

SCA26 The configure operation shall assign values to the properties as indicated in the input configProperties parameter.
3.1.3.2.1.5.5.1.4 Returns
This operation does not return a value.

3.1.3.2.1.5.5.1.5 Exceptions/Errors
SCA27 The configure operation shall raise a PartialConfiguration exception when some configuration properties were successfully set and some configuration properties were not successfully set.
SCA28 The configure operation shall raise an InvalidConfiguration exception when a configuration error occurs and no configuration properties were successfully set.

3.1.3.2.1.5.5.2 query
3.1.3.2.1.5.5.2.1 Brief Rationale
The query operation allows a component to be queried to retrieve its properties.

3.1.3.2.1.5.5.2.2 Synopsis
void query (inout Properties configProperties) raises (UnknownProperties);

3.1.3.2.1.5.5.2.3 Behavior
SCA29 The query operation shall return all component properties when the inout parameter configProperties is zero size. SCA30 The query operation shall return only those id/value pairs specified in the configProperties parameter if the parameter is not zero size.

3.1.3.2.1.5.5.2.4 Returns
This operation does not return a value.

3.1.3.2.1.5.5.2.5 Exceptions/Errors
SCA31 The query operation shall raise the CF::UnknownProperties exception when one or more properties being requested are not known by the component.

3.1.3.2.1.6 ControllableInterface
3.1.3.2.1.6.1 Description
The ControllableInterface interface provides a common API for the control of a software component. The ControllableInterface interface UML is depicted in Figure 3-11.

3.1.3.2.1.6.2 UML

![ControllableInterface Interface UML](image)

**Figure 3-11:** ControllableInterface Interface UML
3.1.3.2.1.6.3 Types
3.1.3.2.1.6.3.1 StartError
The StartError exception indicates that an error occurred during an attempt to start the component. The errorNumber indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```csharp
exception StartError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.2.1.6.3.2 StopError
The StopError exception indicates that an error occurred during an attempt to stop the component. The errorNumber indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```csharp
exception StopError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.2.1.6.4 Attributes
3.1.3.2.1.6.4.1 started
SCA32 The readonly started attribute shall return the component's started value.

```csharp
readonly attribute boolean started;
```

3.1.3.2.1.6.5 Operations
3.1.3.2.1.6.5.1 start
3.1.3.2.1.6.5.1.1 Brief Rationale
The start operation is provided to command the component implementing this interface to start internal processing.

3.1.3.2.1.6.5.1.2 Synopsis
```csharp
void start() raises (StartError);
```

3.1.3.2.1.6.5.1.3 Behavior
The start operation puts the component in an operating condition. The start operation is ignored if the component is already in an operating condition. SCA33 The start operation shall set the started attribute to a value of TRUE.

3.1.3.2.1.6.5.1.4 Returns
This operation does not return a value.

3.1.3.2.1.6.5.1.5 Exceptions/Errors
SCA34 The start operation shall raise the StartError exception if an error occurs while starting the component.

3.1.3.2.1.6.5.2 stop
3.1.3.2.1.6.5.2.1 Brief Rationale
The stop operation is provided to command the component implementing this interface to stop internal processing.

3.1.3.2.1.6.5.2.2 Synopsis
void stop() raises (StopError);

3.1.3.2.1.6.5.2.3 Behavior
The stop operation should disable all current component operations and put it in a non-operating condition. The stop operation is ignored if the component is already in a non-operating condition. SCA36 The stop operation shall set the started attribute to a value of FALSE.

3.1.3.2.1.6.5.2.4 Returns
This operation does not return a value.

3.1.3.2.1.6.5.2.5 Exceptions/Errors
SCA37 The stop operation shall raise the StopError exception if an error occurs while stopping the component.

3.1.3.2.2 Components
Base Application Components provide the structural definitions that will be utilized by application developers.

3.1.3.2.2.1 ApplicationComponent

3.1.3.2.2.1.1 Description
An ApplicationComponent consolidates the common requirements of the SCA Base Application Components. An ApplicationComponent is a constituent part of an AssemblyComponent.

![Application Component UML](image)

**Figure 3-12: Application Component UML**

3.1.3.2.2.1.2 Associations
N/A.

3.1.3.2.2.1.3 Semantics
N/A.

3.1.3.2.2.1.4 Constraints
SCA169 Each ApplicationComponent shall be accompanied by an SPD file per section 3.1.3.6.
SCA173 An ApplicationComponent shall be limited to using the mandatory OS services designated in Appendix B as specified in the SPD.
SCA457 An ApplicationComponent shall be limited to using transfer mechanisms features specified in Appendix E for the specific platform technology implemented. SCA551 An ApplicationComponent shall fulfill the BaseComponent requirements.
3.1.3.2.2.2 ManageableApplicationComponent

3.1.3.2.2.2.1 Description

A ManageableApplicationComponent is a specialization of ApplicationComponent which provides a common API for control and configuration of components within the context of a deployed application.

![ManageableApplicationComponent UML Diagram]

**Figure 3-13: ManageableApplicationComponent UML**

3.1.3.2.2.2.2 Associations

- fileManagement: A ManageableApplicationComponent accesses files via a FileComponent.
- proxyComponent: A ManageableApplicationComponent interfaces with its ApplicationControllerComponent.

3.1.3.2.2.2.3 Semantics

SCA455 Each ManageableApplicationComponent shall support the mandatory Component Identifier execute parameter as described in section 3.1.3.3.1.3.5.1, in addition to their user-defined execute properties in the component’s SPD. SCA168 Each executable ManageableApplicationComponent shall set its identifier attribute using the Component Identifier execute parameter. SCA456 Each executable ManageableApplicationComponent shall accept executable parameters as specified in section 3.1.3.4.1.6.5.1.3 (ExecutableInterface::execute). SCA82 A ManageableApplicationComponent shall register via the ComponentRegistry::registerComponent operation when a COMPONENT_REGISTRY_IOR parameter is supplied. The type field of the registering component is a MANAGEABLE_APPLICATION_COMPONENT.
3.1.3.2.2.2.4 Constraints

SCA520 A ManageableApplicationComponent shall fulfill the ApplicationComponent requirements.

SCA166 A ManageableApplicationComponent shall perform file access through the FileSystem and File interfaces. The application filename syntax is specified in section 3.1.3.5.2.2.3.

SCA167 All ManageableApplicationComponent processes shall have a handler registered for the AEP SIGQUIT signal.

SCA550 A ManageableApplicationComponent shall realize the LifeCycle interface.

3.1.3.2.2.3 ApplicationControllerComponent

3.1.3.2.2.3.1 Description

The ApplicationControllerComponent is the intermediary between the ApplicationManagerComponent and a deployed ManageableApplicationComponent. The ApplicationControllerComponent is the overall controller for an application.

Figure 3-14: ApplicationControllerComponent UML

3.1.3.2.2.3.2 Associations

- appComponent: An ApplicationControllerComponent provides the intermediary between a ManageableApplicationComponent and external entities. Operation invocations, event messages, log messages and exceptions are representative of the type of information that may be exchanged between the components.

3.1.3.2.2.3.3 Semantics

N/A.

3.1.3.2.2.3.4 Constraints

SCA175 An ApplicationControllerComponent shall fulfill the ManageableApplicationComponent requirements. SCA176 An ApplicationControllerComponent shall realize the ControllableInterface interface.
3.1.3.2.2.4 ApplicationComponentFactoryComponent

3.1.3.2.2.4.1 Description
An application component factory is an optional mechanism that may be used exclusively to create application components.

![Diagram of ApplicationComponentFactoryComponent UML](image)

**Figure 3-15: ApplicationComponentFactoryComponent UML**

3.1.3.2.2.4.2 Associations
- createdComponent: An ApplicationComponentFactoryComponent provides a mechanism to create new ApplicationComponents.

3.1.3.2.2.4.3 Semantics
An ApplicationComponentFactoryComponent is used to create an ApplicationComponent. An AssemblyComponent is not required to use an ApplicationComponentFactoryComponent to create application components. A software profile specifies which ApplicationComponentFactoryComponents are to be used by the ApplicationFactoryComponent.

3.1.3.2.2.4.4 Constraints
SCA521 An ApplicationComponentFactoryComponent shall fulfill the BaseFactoryComponent requirements.
SCA522 An ApplicationComponentFactoryComponent shall fulfill the ApplicationComponent requirements.
SCA415 The ApplicationComponentFactoryComponent shall only deploy ApplicationComponents.

3.1.3.2.2.5 AssemblyComponent

3.1.3.2.2.5.1 Description
An AssemblyComponent provides an abstraction for a capability that performs the functions of a specific SCA-compliant product (e.g. a waveform). They are designed to meet the requirements of a specific acquisition and are not defined by the SCA except as they interface to the OE. An AssemblyComponent contains dependencies to BasePlatformComponents, specified as connections, within the descriptor. An instantiated AssemblyComponent contains a collection of ApplicationComponent(s) and optionally nested AssemblyComponent(s).
3.1.3.2.5.2 Associations
- sadProfile: An AssemblyComponent has a SAD and other domain profile files.
- appComponent: The collection of ApplicationComponents identified within the SAD.
- applicationController: The application controller(s) identified within the SAD.
- factoryComponent: The collection of ApplicationComponentFactoryComponents identified within the SAD.
- nestedAssembly: The collection of nested AssemblyComponents identified within the SAD.

3.1.3.2.5.3 Semantics
An AssemblyComponent's dependencies to the log, file system, Event Service, and other ServiceComponents are specified as connections in the SAD using the domainfinder element. Use of an ApplicationComponentFactoryComponent per section 3.1.3.2.4 is optional.

An AssemblyComponent may define interfaces that are visible to entities external to the application. These external interfaces are ports, referenced in the AssemblyComponent SAD externalports element.

3.1.3.2.5.4 Constraints
SCA155 An AssemblyComponent shall be accompanied by the appropriate Domain Profile files per section 3.1.3.6.
SCA156 An AssemblyComponent shall have at least one ApplicationControllerComponent.

3.1.3.3 Framework Control
3.1.3.3.1 Interfaces
Framework control within a Domain is accomplished by domain management and device management interfaces.

The management interfaces are ApplicationManager, DeploymentAttributes, ApplicationFactory, ComponentRegistry, EventChannelRegistry, DomainInstallation, DomainManager, FullComponentRegistry, and ReleasableManager. These interfaces manage the registration,
unregistration, and deployment of applications, devices, and device managers within the domain and control applications within the domain.

Device management is accomplished through the device manager. The device manager creates logical devices and deploys services on these logical devices.

Framework Control Interfaces are implemented using the interface definitions expressed in a Platform Specific representation of one of the Appendix E enabling technologies.

3.1.3.3.1.1 ApplicationManager

3.1.3.3.1.1.1 Description

The ApplicationManager interface provides operations for the control, configuration, and status of instantiated applications within the domain.

The ApplicationManager interface inherits the LifeCycle, ControllableInterface, PropertySet, TestableInterface, and PortAccessor interfaces. The ApplicationManager interface UML is depicted in Figure 3-17.

The ApplicationManager::releaseObject operation provides the interface to release the computing resources allocated during the instantiation of the application, and de-allocate the devices associated with ApplicationManager instance.

3.1.3.3.1.1.2 UML

Figure 3-17: ApplicationManager Interface UML

3.1.3.3.1.1.3 Types

N/A.

3.1.3.3.1.1.4 Attributes

3.1.3.3.1.1.4.1 name

SCA41 The readonly name attribute shall return the name of the created application. The ApplicationFactory interface’s create operation name parameter provides the name content.

readonly attribute string name;
3.1.3.3.1.1.5 Operations

3.1.3.3.1.1.5.1 releaseObject

3.1.3.3.1.1.5.1.1 Brief Rationale

The `releaseObject` operation terminates execution of the application, returns all allocated computing resources, and de-allocates the resources' capacities in use by the devices associated with the application. Before termination, the application removes the message connectivity with its associated applications (e.g. port to port connections) in the domain.

3.1.3.3.1.1.5.1.2 Synopsis

```c
void releaseObject() raises (ReleaseError);
```

3.1.3.3.1.1.5.1.3 Behavior

The following behavior extends the `LifeCycle::releaseObject` operation requirements (see section 3.1.3.2.1.3.5.2)

SCA42 The `ApplicationManager::releaseObject` operation shall release each application component by utilizing the `LifeCycle::releaseObject` operation. SCA43 The `ApplicationManager::releaseObject` operation shall terminate the processes / tasks on allocated ExecutableDeviceComponents belonging to each application component.

SCA44 The `ApplicationManager::releaseObject` operation shall unload each application component instance from its allocated LoadableDeviceComponent.

SCA45 The `ApplicationManager::releaseObject` operation shall deallocate the DeviceComponent capacities that were allocated during application creation.

SCA46 The `ApplicationManager::releaseObject` operation shall release all object references to the components making up the application.

SCA47 The `ApplicationManager::releaseObject` operation shall disconnect ports (including an Event Service's event channel consumers and producers) that were previously connected based upon the application's associated SAD. The `ApplicationManager::releaseObject` operation may destroy an Event Service's event channel when no more consumers and producers are connected to it.

The `ApplicationManager::releaseObject` operation for an application should disconnect ports first, then release its components, call the `terminate` operation, and lastly call the `unload` operation on the DeviceComponents.

SCA49 The `ApplicationManager::releaseObject` operation shall, upon successful application release, write an ADMINISTRATIVE_EVENT log record.

SCA50 The `ApplicationManager::releaseObject` operation shall, upon unsuccessful application release, write a FAILURE_ALARM log record.

SCA51 The `ApplicationManager::releaseObject` operation shall send a ComponentChangeEvent Type event to the Outgoing Domain Management event channel upon successful release of an application. For this event:

1. The producerId is the identifier attribute of the releasing ApplicationManagerComponent;
2. The componentChange is REMOVED;
3. The **domainComponent** is the released ApplicationManagerComponent's CF::ComponentType.  

The following steps demonstrate one scenario of the application's behavior for the release of an application:

1. Client invokes **ApplicationManager::releaseObject** operation;
2. Disconnect ports to application and platform components based upon the SAD;
3. Release the application components;
4. Terminate the application components' and component factories processes;
5. Unload the components' executable images;
6. Deallocate capacities based upon the Device Profile and SAD;
7. Unregister application components from the component registry;
8. Generate an event to indicate the application has been removed from the domain;

Figure 3-18 is a sequence diagram depicting the behavior as described above.

![Sequence Diagram](image)

**Figure 3-18:** **ApplicationManager** Behavior

3.1.3.3.1.5.1.4 Returns
This operation does not return a value.

3.1.3.3.1.5.1.5 Exceptions/Errors
The `ApplicationManager::releaseObject` operation raises a `ReleaseError` exception when internal processing errors prevent the successful release of any application component. See section 3.1.3.2.1.3.5.2.5 for exception handling.

3.1.3.3.1.1.5.2 getProvidesPorts

3.1.3.3.1.1.5.2.1 Brief Rationale

The `getProvidesPorts` operation is used to retrieve the application's external provides ports as defined in the associated SAD (ApplicationManagerComponent's profile). This operation overrides the definition in `PortAccessor::getProvidesPorts` section 3.1.3.2.1.2.5.3.

3.1.3.3.1.1.5.2.2 Synopsis

```c
void getProvidesPorts (inout Connections portConnections) raises (InvalidPort);
```

3.1.3.3.1.1.5.2.3 Behavior

The `getProvidesPorts` operation returns the external provides port object references for the external provides ports as stated in the associated SAD (ApplicationManagerComponent's profile).

3.1.3.3.1.1.5.2.4 Returns

SCA53 The `getProvidesPorts` operation shall return the object references that are associated with the input provides port names for the application external ports as identified in the associated SAD (ApplicationManagerComponent's profile).

3.1.3.3.1.1.5.2.5 Exceptions/Errors

Exception requirement(s) are described in the `PortAccessor` interface section 3.1.3.2.1.2.5.3.5.

3.1.3.3.1.1.5.3 connectUsesPorts

3.1.3.3.1.1.5.3.1 Brief Rationale

The `connectUsesPorts` operation is used to connect to the application external uses ports as defined in the associated SAD (profile attribute). This operation overrides the definition in `PortAccessor::connectUsesPorts` section 3.1.3.2.1.2.5.1.

3.1.3.3.1.1.5.3.2 Synopsis

```c
void connectUsesPorts (in Connections portConnections) raises (InvalidPort);
```

3.1.3.3.1.1.5.3.3 Behavior

SCA55 The `connectUsesPorts` operation shall make a connection to the application components by input `portConnections` parameter, which identifies the application external uses ports to be connected to. Application external ports are associated with the application components.

SCA523 The `connectUsesPorts` operation shall disconnect any connections it formed if any connections in the input `portConnections` parameter cannot be successfully established.

3.1.3.3.1.1.5.3.4 Returns

This operation does not return a value.

3.1.3.3.1.1.5.3.5 Exceptions/Errors

Exception requirement(s) are described in the `PortAccessor` interface section 3.1.3.2.1.2.5.1.5.
3.1.3.3.1.5.4 disconnectPorts

3.1.3.3.1.5.4.1 Brief Rationale

The disconnectPorts operation is used to disconnect the application external ports as defined in the associated SAD (ApplicationManagerComponent's profile). This operation overrides the definition in PortAccessor::disconnectPorts section 3.1.3.2.1.2.5.2.

3.1.3.3.1.5.4.2 Synopsis

```c
void disconnectPorts (in Disconnections portDisconnections)
raises (InvalidPort);
```

3.1.3.3.1.5.4.3 Behavior

SCA58 The disconnectPorts operation shall break the connection(s) to the application external ports as identified by the connectionIds referenced in the input portDisconnections parameter.

SCA59 The disconnectPorts operation shall release all external ports if the input portDisconnections parameter is a zero length sequence.

3.1.3.3.1.5.4.4 Returns

This operation does not return a value.

3.1.3.3.1.5.4.5 Exceptions/Errors

Exception requirement(s) are described in the PortAccessor interface section 3.1.3.2.1.2.5.2.5.

3.1.3.3.1.2 DeploymentAttributes

3.1.3.3.1.2.1 Description

The DeploymentAttributes interface provides deployment attributes for an application.

For each deployed BaseComponent, the deployed component specializedInfo contains deployment information consisting of:

1. A CF::DataType with an id of EXECUTION_ID and value of ExecutableInterface::ExecutionID_Type;
2. A CF::DataType with an id of IMPLEMENTATION_ID and value of the profile's implementation id used for deployment;
3. A CF::DataType with an id of TARGET_DEVICE_ID and value of the DeviceComponent's identifier that component is deployed on;
4. A CF::DataType with an id of USES_DEVICE_ID and a value of CF::UsesDeviceAssignmentSequence type if a component has uses devices as indicated by its profile.

3.1.3.3.1.2.2 UML

![DeploymentAttributes Interface UML](image)

Figure 3-19: DeploymentAttributes Interface UML
3.1.3.3.1.2.3 Types
3.1.3.3.1.2.4 Attributes
3.1.3.3.1.2.4.1 deployedComponents
SCA64 The deployedComponents attribute shall return the list of BaseComponents that have been successfully deployed or a sequence length of zero if no BaseComponents have been deployed.

readonly attribute Components deployedComponents;

3.1.3.3.1.2.5 Operations
N/A.

3.1.3.3.1.3 ApplicationFactory
3.1.3.3.1.3.1 Description
The ApplicationFactory interface class provides an interface to request the creation of a specific application in the domain.

The ApplicationFactory interface class is designed using the Factory Design Pattern [10].

3.1.3.3.1.3.3 UML

![UML diagram of ApplicationFactory interface]

Figure 3-20: ApplicationFactory Interface UML

3.1.3.3.1.3.3 Types
3.1.3.3.1.3.3.1 CreateApplicationRequestError Exception
The CreateApplicationRequestError exception is raised when the parameter CF::DeviceAssignmentSequence contains one or more invalid application component-to-device assignment(s).

```
exception CreateApplicationRequestError {
    DeviceAssignmentSequence invalidAssignment;
};
```

3.1.3.3.1.3.3.2 CreateApplicationError Exception
The CreateApplicationError exception is raised when a create request is valid but the application is unsuccessfully instantiated. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```
exception CreateApplicationError { ErrorNumberType errorNumber;
    string msg;
};
```
3.1.3.3.1.3.3.3 Exception InvalidInitConfiguration
The InvalidInitConfiguration exception is raised when the input initConfiguration parameter is
invalid.

exception InvalidInitConfiguration { Properties
invalidProperties; };

3.1.3.3.1.3.3.4 ExecutionAffinityType
The ExecutionAffinityType structure associates a component with a process collocation and/or
processor core on which it is executed. Only processCollocation and coreAffinity values that are
specified (non-empty) are used.

struct ExecutionAffinityType
{
    string componentId;
    string processCollocation;
    CF::ULongSeq coreAffinities;
};

3.1.3.3.1.3.3.5 ExecutionAffinitySequence
The ExecutionAffinitySequence sequence provides an unbounded sequence of

ExecutionAffinityTypes.

typedef sequence <ExecutionAffinityType>
ExecutionAffinitySequence;

3.1.3.3.1.3.4 Attributes
3.1.3.3.1.3.4.1 name
SCA65 The readonly name attribute shall return the name of the application instantiated by an
application factory. The name attribute is identical to the softwareassembly element name
attribute of the application's SAD file.

readonly attribute string name;

3.1.3.3.1.3.5 Operations
3.1.3.3.1.3.5.1 create
3.1.3.3.1.3.5.1.1 Brief Rationale
The create operation is used to create an application within the system domain.
The create operation provides a client interface to request the creation of an application on client
requested device(s) and/or the creation of an application in which the application factory
determines the necessary device(s) required for instantiation of the application.

3.1.3.3.1.3.5.1.2 Synopsis

ComponentType create (in string name, in Properties
initConfiguration, in DeviceAssignmentSequence
deviceAssignments, in Properties deploymentDependencies, in
ExecutionAffinitySequence executionAffinityAssignments) raises
(CreateApplicationError, CreateApplicationRequestError,
InvalidInitConfiguration);
### 3.1.3.3.1.3.5.1.3 Behavior

The *create* operation locates candidate DomainManagerComponents, ApplicationFactoryComponents or DeviceComponents capable of deploying application software modules based upon information in its associated SAD (ApplicationFactoryComponent's profile).

The *create* operation validates all component-device associations in the input deviceAssignments parameter by verifying that the DeviceComponent indicated by the assignedDeviceId element provides the necessary capacities and properties required by the component indicated by the componentId element.

**SCA69** The *create* operation shall use the allocation property values contained in the input deploymentDependencies parameter over the application deploymentdependencies elements or components dependency allocation properties of application factory profile when they reference the same property.

**SCA575** The *create* operation shall use the affinity values contained in the input executionAffinityAssignments parameter prior to those specified by the ApplicationFactoryComponent profile's processcollocation attribute and/or coreaffinity element.

**SCA70** The *create* operation shall pass the input deploymentDependencies parameter for nested assemblyinstantiation elements creation.

The *create* operation may also use the input deploymentDependencies parameter for other deployment decisions.

The *create* operation ignores input deploymentDependencies parameter properties that are unknown.

**SCA74** The create operation shall deploy the ApplicationComponents as specified in the SAD.

**SCA75** The create operation shall use each component's SPD implementation code's stacksize and priority elements, when specified, for the execute options parameters.

**SCA84** The *create* operation shall, in order, initialize all ApplicationComponents, establish connections for those components, and finally configure ManageableApplicationComponent(s) as identified by the assemblycontroller element in the SAD. The *create* operation connects the ports of the application components with the ports of other components within the application as well as the devices and services they use in accordance with the SAD.

**SCA91** The *create* operation shall use the property values contained in the input initConfiguration parameter over the property values of the SAD's assemblycontroller element when they reference the same property.

**SCA92** The *create* operation shall recognize application deployment channel preferences contained within an ADD file.

**SCA93** The *create* operation shall recognize a deploymentDependencies property which is a CF::Properties type with an id of "DEPLOYMENT_CHANNEL" and a value that is a string sequence.

**SCA94** The *create* operation shall recognize channel preferences contained within a "DEPLOYMENT_CHANNEL" deploymentDependency property contained within the deploymentDependencies parameter.
SCA95 The create operation shall attempt to allocate an application to the PDD file channel alternatives provided within a "DEPLOYMENT_CHANNEL" property or an ADD file in a sequential manner.

SCA96 The create operation shall utilize channel preferences expressed within a "DEPLOYMENT_CHANNEL" property rather than those contained within an ADD file if both exist.

SCA97 The create operation shall recognize a deployment option with a deployedname attribute value of "DEFAULT" which matches all application instance names that are not explicitly identified by a deployedname attribute value within the same descriptor file.

The TestableInterface::runTest operation (3.1.3.2.1.4.5.1), ControllableInterface::stop operation (3.1.3.2.1.6.5.2), and ControllableInterface::start operation (3.1.3.2.1.6.5.1) are not called as part of the application creation process.

3.1.3.3.1.3.5.1.4 Returns

SCA102 The create operation shall return the created ApplicationManagerComponent's CF::ComponentType for the created application when the application is successfully created.

SCA576 The create operation's returned CF::ComponentType's specializedInfo shall contain the application's deployed components as identified by COMPONENTS_ID and CF::Components type value. The returned CF::ComponentType's identifier field is the input name parameter. The returned CF::ComponentType's providesPorts field is the ApplicationManagerComponent's external provides ports. For each deployed component, it's CF::ComponentType's identifier, componentObject, profile, providesPorts, and type fields are as specified in section 3.1.3.1.3.17, CF::ComponentType.

Each application deployed component's specializedInfo contains deployment information consisting of:

1. A CF::DataType with an id of EXECUTION_ID and value of ExecutableInterface::ExecutionID_Type;
2. A CF::DataType with an id of IMPLEMENTATION_ID and value of the profile's implementation id used for deployment;
3. A CF::DataType with an id of TARGET_DEVICE_ID and value of the DeviceComponent's identifier that the component is deployed on;
4. A CF::DataType with an id of USES_DEVICE_ID and a value of CF::UsesDeviceAssignmentSequence type when a component has uses devices as indicated by its profile.

3.1.3.3.1.3.5.1.5 Exceptions/Errors

SCA103 The create operation shall raise the CreateApplicationRequestError exception when the input deviceAssignments parameter contains one or more invalid application component to device assignment(s).

SCA104 The create operation shall raise the CreateApplicationError exception when the create request is valid but the application cannot be successfully instantiated due to internal processing error(s).
SCA105 The `create` operation shall raise the CreateApplicationError exception when the CF implementation provides enhanced deployment support via the use of a PDD file if the CF is not able to allocate the application to any of the provided channel alternatives.

SCA106 The `create` operation shall raise the CreateApplicationError exception when the CF implementation provides enhanced deployment support via the use of a PDD file and a domainfinder element "servicetype" connection to a ServiceComponent whose service type is provided by a service contained within a channel element servicelist cannot be established to a service identified within that list.

SCA570 The `create` operation shall raise the CreateApplicationError exception when an ApplicationManagerComponent already exists in the system with a CF::ComponentType identifier attribute value equal to that of the input name parameter.

SCA107 The `create` operation shall raise the InvalidInitConfiguration exception when the input initConfiguration parameter contains properties that are unknown by a SAD's assemblycontroller element. SCA108 The InvalidInitConfiguration invalidProperties parameter shall identify the invalid properties.

3.1.3.3.1.4 DomainManager

3.1.3.3.1.4.1 Description

The DomainManager interface operations are used to configure the domain and manage the domain's devices, services, and applications.

3.1.3.3.1.4.2 UML

The DomainManager interface UML is depicted in Figure 3-21.

![Figure 3-21: DomainManager Interface UML](image)

3.1.3.3.1.4.3 Types

N/A.

3.1.3.3.1.4.4 Attributes

3.1.3.3.1.4.4.1 managers

The managers attribute is read-only, containing a sequence of registered DeviceManagerComponents in the domain. SCA109 The readonly managers attribute shall
return a list of DeviceManagerComponents that have registered with the DomainManagerComponent.

readonly attribute Components managers;

3.1.3.3.1.4.4.2 applications

The applications attribute is read-only containing a sequence of ApplicationManagerComponents in the domain. SCA110 The readonly applications attribute shall return the list of ApplicationManagerComponents that have been instantiated.

readonly attribute Components applications;

3.1.3.3.1.4.4.3 applicationFactories

SCA435 The readonly applicationFactories attribute shall return a list with one ApplicationFactoryComponent per AssemblyComponent (SAD file and associated files) successfully installed (i.e. no exception raised).

readonly attribute Components applicationFactories;

3.1.3.3.1.4.4.4 fileMgr

SCA111 The readonly fileMgr attribute shall return the DomainManagerComponent's FileManagerComponent.

readonly attribute FileManager fileMgr;

3.1.3.3.1.4.4.5 domainManagerProfile

SCA112 The readonly domainManagerProfile attribute shall return the filename of the DomainManagerComponent's DMD or the DMD itself. The filename is an absolute pathname relative to a mounted FileSystemComponent and the file is obtained via the DomainManagerComponent's FileManagerComponent. Files referenced within the profile are obtained via the DomainManagerComponent's FileManagerComponent.

readonly attribute string domainManagerProfile;

3.1.3.3.1.4.5 Operations

N/A.

3.1.3.3.1.5 DomainInstallation

3.1.3.3.1.5.1 Description

The DomainInstallation interface is used for the control of application installation within the system domain.

3.1.3.3.1.5.2 UML

The DomainInstallation interface UML is depicted in Figure 3-22.

![DomainInstallation Interface UML](image)

**Figure 3-22: DomainInstallation Interface UML**
3.1.3.3.1.5.3 Types

3.1.3.3.1.5.3.1 ApplicationInstallationError

The ApplicationInstallationError exception type is raised when an application installation has not completed correctly. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```
exception ApplicationInstallationError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.3.1.5.3.2 InvalidIdentifier

The InvalidIdentifier exception indicates an application identifier is invalid.
```
exception InvalidIdentifier{}
```

3.1.3.3.1.5.3.3 ApplicationUninstallationError

The ApplicationUninstallationError exception type is raised when the uninstallation of an application has not completed correctly. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```
exception ApplicationUninstallationError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.3.1.5.3.4 ApplicationAlreadyInstalled

The ApplicationAlreadyInstalled exception indicates that the application being installed is already installed.
```
exception ApplicationAlreadyInstalled{}
```

3.1.3.3.1.5.4 Attributes.

N/A.

3.1.3.3.1.5.5 Operations

3.1.3.3.1.5.5.1 installApplication

3.1.3.3.1.5.5.1.1 Brief Rationale

The installApplication operation is used to install new application software in the domain.

3.1.3.3.1.5.5.1.2 Synopsis

```
ComponentType installApplication (in string profileFileName) raises (InvalidProfile, InvalidFileName, ApplicationInstallationError, ApplicationAlreadyInstalled);
```

3.1.3.3.1.5.5.1.3 Behavior

The input profileFileName parameter is the absolute pathname of the AssemblyComponent's SAD.

SCA113 The installApplication operation shall verify the existence of the AssemblyComponent's SAD file and all files upon which the SAD depends, within the DomainManagerComponent's file manager.

SCA114 The installApplication operation shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log, upon successful application installation.
SCA Specification

Version: 4.1
20 August 2015

SCA115 The `installApplication` operation shall, upon unsuccessful application installation, write a FAILURE_ALARM log record to a DomainManagerComponent's log.

SCA116 The `installApplication` operation shall send a ComponentChangeEvent: event to the Outgoing Domain Management event channel, upon successful installation of an application. For this event:

1. The `producerId` is the identifier attribute of the DomainManagerComponent;
2. The `componentChange` is ADDED;
3. The `domainComponent` is the installed ApplicationFactoryComponent's CF::ComponentType.

3.1.3.3.1.5.5.1.4 Returns

SCA571 The `installApplication` operation shall return the installed ApplicationFactoryComponent's CF::ComponentType. For the returned CF::ComponentType, its profile, componentObject, and type fields are as specified in section 3.1.3.1.3.17, CF::ComponentType. The returned CF::ComponentType's identifier field is the input profileFileName's element name attribute.

3.1.3.3.1.5.5.1.5 Exceptions/Errors

SCA117 The `installApplication` operation shall raise the ApplicationInstallationError exception when the installation of the application file(s) was not successfully completed.

SCA118 The `installApplication` operation shall raise the CF::InvalidFileName exception when the input SAD file or any of the SAD's referenced filenames do not exist in the file system identified by the absolute path of the input profileFileName parameter.

SCA119 The `installApplication` operation shall log a FAILURE_ALARM log record to a DomainManagerComponent's Log with a message consisting of "installApplication::invalid file is xxx", where "xxx" is the input or referenced filename, when the CF InvalidFileName exception occurs.

SCA120 The `installApplication` operation shall raise the CF::InvalidProfile exception when any referenced property definition is missing.

SCA121 The `installApplication` operation shall write a FAILURE_ALARM log record to a DomainManagerComponent's log when the CF::InvalidProfile exception is raised. The value of the logData attribute of this record is "installApplication::invalid Profile is yyy", where "yyy" is the input or referenced file name.

SCA122 The `installApplication` operation shall raise the ApplicationAlreadyInstalled exception when the softwareassembly element name attribute of the referenced application is the same as a previously registered application.

3.1.3.3.1.5.5.2 uninstallApplication

3.1.3.3.1.5.5.2.1 Brief Rationale

The `uninstallApplication` operation is used to uninstall an application factory from the domain.

3.1.3.3.1.5.5.2.2 Synopsis

```c
void uninstallApplication (in string identifier)raises
(InvalidIdentifier, ApplicationUninstallationError);
```
3.1.3.3.1.5.5.2.3 Behavior

The identifier parameter is the `softwareassembly` element `name` attribute of the AssemblyComponent's SAD file.

SCA436 The `uninstallApplication` operation shall make the ApplicationFactoryComponent unavailable from the DomainManagerComponent (i.e. its services no longer provided for the application).

SCA123 The `uninstallApplication` operation shall, upon successful uninstall of an application, write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log.

SCA124 The `uninstallApplication` operation shall, upon unsuccessful uninstall of an application, write a FAILURE_ALARM log record to a DomainManagerComponent's log.

SCA125 The `uninstallApplication` operation shall send a ComponentChangeEvent`EventType` event to the Outgoing Domain Management event channel, upon the successful uninstallation of an application. For this event:

1. The `producerId` is the identifier attribute of the DomainManagerComponent;
2. The `componentChange` is REMOVED;
3. The `domainComponent` is the uninstalled ApplicationFactoryComponent's CF::ComponentType.

3.1.3.3.1.5.5.2.4 Returns

This operation does not return a value.

3.1.3.3.1.5.5.2.5 Exceptions/Errors

SCA126 The `uninstallApplication` operation shall raise the InvalidIdentifier exception when the identifier parameter is invalid.

SCA127 The `uninstallApplication` operation shall raise the ApplicationUninstallationError exception when an internal error causes an unsuccessful uninstallation of the application.

3.1.3.3.1.6 ComponentRegistry

3.1.3.3.1.6.1 Description

The `ComponentRegistry` interface is used to manage the registration of components.

3.1.3.3.1.6.2 UML

![ComponentRegistry UML Diagram]

Figure 3-23: ComponentRegistry Interface UML
3.1.3.3.1.6.3 Types

3.1.3.3.1.6.3.1 RegisterError

The RegisterError exception indicates that an internal error has occurred which prevents the ComponentRegistry interface registration operations from successful completion. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```plaintext
exception RegisterError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.3.1.6.4 Attributes

N/A.

3.1.3.3.1.6.5 Operations

3.1.3.3.1.6.5.1 registerComponent

3.1.3.3.1.6.5.1.1 Brief Rationale

This registerComponent operation registers a component and its provides ports.

3.1.3.3.1.6.5.1.2 Synopsis

```plaintext
void registerComponent (in ComponentType registeringComponent) raises (InvalidObjectReference, RegisterError);
```

3.1.3.3.1.6.5.1.3 Behavior

The registerComponent operation verifies that the input registeringComponent parameter contains a valid component reference.

SCA131 The registerComponent operation shall register the component indicated by the input registeringComponent parameter, if it does not already exist.

The registerComponent operation ignores already existing registrations.

3.1.3.3.1.6.5.1.4 Returns

This operation does not return any value.

3.1.3.3.1.6.5.1.5 Exceptions/Errors

SCA132 The registerComponent operation shall raise the CF::InvalidObjectReference when the input registeringComponent contains a nil componentObject object reference.

SCA133 The registerComponent operation shall raise the RegisterError exception when registration is unsuccessful.

3.1.3.3.1.7 FullComponentRegistry

3.1.3.3.1.7.1 Description

The FullComponentRegistry interface extends the ComponentRegistry interface with unregistration capability.
3.1.3.3.1.7.2 UML

![UML Diagram]

*Figure 3-24: FullComponentRegistry Interface UML*

3.1.3.3.1.7.3 Types
3.1.3.3.1.7.3.1 UnregisterError

The UnregisterError exception indicates that an internal error has occurred which prevents the `FullComponentRegistry` interface unregister operations from successful completion. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```cpp
exception UnregisterError { ErrorNumberType errorNumber; string msg; }
```

3.1.3.3.1.7.4 Attributes

N/A.

3.1.3.3.1.7.5 Operations
3.1.3.3.1.7.5.1 unregisterComponent
3.1.3.3.1.7.5.1.1 Brief Rationale

The `unregisterComponent` operation unregisters the component as identified by the input identifier parameter.

3.1.3.3.1.7.5.1.2 Synopsis

```cpp
void unregisterComponent (in string identifier) raises (UnregisterError);
```

3.1.3.3.1.7.5.1.3 Behavior

SCA134 The `unregisterComponent` operation shall unregister a registered component entry specified by the input identifier parameter.

3.1.3.3.1.7.5.1.4 Returns

This operation does not return any value.

3.1.3.3.1.7.5.1.5 Exceptions/Errors
SCA135 The `unregisterComponent` operation shall raise the UnregisterError exception when unregistration is unsuccessful.

3.1.3.3.1.8 EventChannelRegistry

3.1.3.3.1.8.1 Description

The `EventChannelRegistry` interface is used to manage the registration processes with the event channel.

3.1.3.3.1.8.2 UML

The `EventChannelRegistry` interface UML is depicted in Figure 3-25.

```plaintext
+ registerWithEventChannel(Object, string, string): void
+ unregisterFromEventChannel(string, string): void
```

**Figure 3-25: EventChannelRegistry Interface UML**

3.1.3.3.1.8.3 Types

3.1.3.3.1.8.3.1 InvalidEventChannelName

The `InvalidEventChannelName` exception indicates an event channel name that is unknown.

```plaintext
exception InvalidEventChannelName{};
```

3.1.3.3.1.8.3.2 AlreadyConnected

The `AlreadyConnected` exception indicates that a registering consumer is already connected to the specified event channel.

```plaintext
exception AlreadyConnected{};
```

3.1.3.3.1.8.3.3 NotConnected

The `NotConnected` exception indicates that the unregistering consumer was not connected to the specified event channel.

```plaintext
exception NotConnected{};
```

3.1.3.3.1.8.4 Attributes.

N/A.

3.1.3.3.1.8.5 Operations

3.1.3.3.1.8.5.1 registerWithEventChannel

3.1.3.3.1.8.5.1.1 Brief Rationale

The `registerWithEventChannel` operation is used to connect a consumer to a domain's event channel.

3.1.3.3.1.8.5.1.2 Synopsis
void registerWithEventChannel (in Object registeringObject, in string registeringId, in string eventChannelName) raises (InvalidObjectReference, InvalidEventChannelName, AlreadyConnected);

3.1.3.3.1.8.5.1.3 Behavior
SCA136 The registerWithEventChannel operation shall connect, with a connection named by the input registeringId parameter, the object contained within the input registeringObject parameter to an event channel specified by the input eventChannelName parameter.

3.1.3.3.1.8.5.1.4 Returns
This operation does not return a value.

3.1.3.3.1.8.5.1.5 Exceptions/Errors
SCA137 The registerWithEventChannel operation shall raise the CF::InvalidObjectReference exception when the input registeringObject parameter contains an invalid reference to a CosEventComm::PushConsumer interface.
SCA138 The registerWithEventChannel operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter contains an invalid event channel name.
SCA139 The registerWithEventChannel operation shall raise AlreadyConnected exception when the object contained within the input registeringObject parameter already contains a connection identified by the input registeringId parameter.

3.1.3.3.1.8.5.2 unregisterFromEventChannel

3.1.3.3.1.8.5.2.1 Brief Rationale
The unregisterFromEventChannel operation is used to disconnect a consumer from a domain's event channel.

3.1.3.3.1.8.5.2.2 Synopsis
void unregisterFromEventChannel (in string unregisteringId, in string eventChannelName) raises (InvalidEventChannelName, NotConnected);

3.1.3.3.1.8.5.2.3 Behavior
SCA140 The unregisterFromEventChannel operation shall disconnect a registered component from the event channel as identified by the input parameters.

3.1.3.3.1.8.5.2.4 Returns
This operation does not return a value.

3.1.3.3.1.8.5.2.5 Exceptions/Errors
SCA141 The unregisterFromEventChannel operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter can't be located as a named event channel within the domain.
SCA142 The unregisterFromEventChannel operation shall raise the NotConnected exception when the input unregisteringId parameter is not associated with the input eventChannelName parameter.
3.1.3.3.1.9 ReleasableManager
3.1.3.3.1.9.1 Description
The ReleasableManager interface is used for terminating a manager.
3.1.3.3.1.9.2 UML

```uml
<<interface>>
CF::ReleaseableManager
+ shutdown(): void
```

**Figure 3-26: ReleasableManager Interface UML**

3.1.3.3.1.9.3 Types
N/A.
3.1.3.3.1.9.4 Attributes
N/A.
3.1.3.3.1.9.5 Operations
3.1.3.3.1.9.5.1 shutdown
3.1.3.3.1.9.5.1.1 Brief Rationale
The shutdown operation provides the mechanism to terminate a manager.
3.1.3.3.1.9.5.1.2 Synopsis
void shutdown();
3.1.3.3.1.9.5.1.3 Behavior
SCA151 The shutdown operation shall unregister the manager from the domain.
SCA152 The shutdown operation shall perform a releaseObject on all of the manager's registered components that support the LifeCycle interface.
SCA153 The shutdown operation shall terminate the execution of each component created as specified in the manager's profile after it has unregistered from the manager.
SCA437 The shutdown operation shall cause the manager to be unavailable (i.e. released from the operating environment and its process terminated on the OS), when all of the manager's registered components are unregistered and all created components are terminated.
3.1.3.3.1.9.5.1.4 Returns
This operation does not return any value.
3.1.3.3.1.9.5.1.5 Exceptions/Errors
This operation does not raise any exceptions.
3.1.3.3.2 Components

Framework Control Components provide the structural definitions for the components that perform deployment behavior within a platform. Framework control within a Domain is accomplished by the DomainManagerComponent and DeviceManagerComponent.

All Framework Control Components provide a management capability. These components manage the registration and unregistration of applications, devices, and device managers within the domain and the controlling of applications within the domain. The implementation of the ApplicationManagerComponent, ApplicationFactoryComponent, and DomainManagerComponent components are logically coupled to provide a complete domain management implementation and service.

BasePlatformComponent management is performed by the DeviceManagerComponent. The DeviceManagerComponent is responsible for deploying DeviceComponents and ServiceComponents.

3.1.3.3.2.1 ApplicationManagerComponent

3.1.3.3.2.1.1 Description

The ApplicationManagerComponent provides the means for the control, configuration, and status of an AssemblyComponent in the domain. The ApplicationManagerComponent is the proxy for the deployed AssemblyComponent.

An ApplicationManagerComponent is returned by the *create* operation of an ApplicationFactoryComponent.
3.1.3.3.2.1.2 Associations

- **eventChannel**: An ApplicationManagerComponent sends event messages to event channels, disconnects producers and consumers from event channels and may destroy event channels.
- **capacityProvider**: An ApplicationManagerComponent deallocates capacities from its managed components from the DeviceComponent(s) upon which they are deployed.
- **targetLog**: An ApplicationManagerComponent produces log messages and disseminates them to system log(s).
- **appController**: An ApplicationManagerComponent delegates requests for an AssemblyComponent to its ApplicationControllerComponent.
- **componentContainer**: An ApplicationManagerComponent releases ManageableApplicationComponents using an ApplicationComponentFactoryComponent(s).
- **processContainer**: An ApplicationManagerComponent terminates its processes from their containing ExecutableDeviceComponent(s).
- **moduleContainer**: An ApplicationManagerComponent unloads its constituent software modules from their containing LoadableDeviceComponent(s).
• component: An ApplicationManager manages connections to other component types (e.g., ManageableApplicationComponent, BasePlatformComponent, or ApplicationManagerComponent) with which an instantiated AssemblyComponent has been connected.

3.1.3.3.2.1.3 Semantics

SCA158 An ApplicationManagerComponent shall delegate the runTest, start, stop, configure, and query operations to the ApplicationControllerComponent(s) as identified by the AssemblyComponent's SAD assemblycontroller element (application controller).

SCA159 The ApplicationManagerComponent shall propagate exceptions raised by the AssemblyComponent's ApplicationControllerComponent(s).

SCA160 The ApplicationManagerComponent shall not delegate the initialize operation to its ApplicationComponentFactoryComponent(s), ManageableApplicationComponent(s) or ApplicationControllerComponent(s).

SCA161 The ApplicationManagerComponent shall delegate the runTest operation to all component(s) as identified by the AssemblyComponent's SAD assemblycontroller element (application controller) which have matching test IDs.

SCA162 The ApplicationManagerComponent shall delegate configure and query operations to all ManageableApplicationComponent(s) as identified by the AssemblyComponent's SAD assemblycontroller element (application controller), which have matching property IDs.

SCA163 The ApplicationManagerComponent shall raise the configure operation's InvalidConfiguration exception when the input configProperties parameter contains unknown properties. A property is considered unknown if it can't be recognized by a component designated by the ApplicationManagerComponent profile's assemblycontroller element.

SCA543 The ApplicationManagerComponent shall raise the query operation's UnknownProperties exception when the input configProperties parameter contains unknown properties.

3.1.3.3.2.1.4 Constraints

SCA164 An ApplicationManagerComponent shall realize the ApplicationManager interface.

SCA165 An ApplicationManagerComponent shall fulfill the BaseComponent requirements.

3.1.3.3.2.2 ApplicationFactoryComponent

3.1.3.3.2.2.1 Description

The ApplicationFactoryComponent provides a dynamic mechanism to create a specific type of ApplicationManagerComponent in the domain.
3.1.3.2.2.2 Associations

- **domainProfile:** An ApplicationFactoryComponent is associated with a SAD and zero to many other domain profile files.
- **eventChannel:** An ApplicationFactoryComponent sends event messages to event channels, disconnects producers and consumers from event channels and may destroy event channels.
- **targetLog:** An ApplicationFactoryComponent produces log messages and disseminates them to system log(s).
- **processContainer:** An ApplicationFactoryComponent initiates processes on ExecutableDeviceComponent(s).
- **moduleContainer:** An ApplicationFactoryComponent loads modules onto LoadableDeviceComponent(s).
- **capacityProvider:** An ApplicationFactoryComponent allocates DeviceComponent(s) capacities.
- **appComponent:** An ApplicationFactoryComponent initializes, configures and manages connections for its ApplicationComponent(s).
- **componentContainer:** An ApplicationFactoryComponent creates ApplicationComponents via ApplicationComponentFactoryComponent(s).
- **appManager:** An ApplicationFactoryComponent creates an ApplicationManagerComponent which acts as a proxy for the instantiated AssemblyComponent.
• componentRegistry: An ApplicationFactoryComponent obtains componentRegistries that contain an inventory of the created application component(s).
• component: An ApplicationFactoryComponent connects deployed application components.

3.1.3.3.2.2.3 Semantics

SCA68 The create operation shall identify valid component-device associations for the application by matching the allocation properties of the application to those of each candidate DeviceComponent, for those ManageableApplicationComponent properties whose kindtype is "allocation" and whose action element is not "external".

SCA71 The create operation shall allocate capacities to candidate DeviceComponents of the ApplicationComponent properties whose kindtype is "allocation" and whose action element is "external".

SCA72 The create operation shall deallocate any capacity allocations on DeviceComponents that do not satisfy the ApplicationComponent's allocation requirements or that are not utilized due to an unsuccessful application creation.

SCA73 The create operation shall load application modules onto DeviceComponents that have been granted successful capacity allocations and satisfy the ApplicationComponent's allocation requirements.

SCA76 When the create operation deploys an ApplicationComponent via an ExecutableDeviceComponent, it shall include a Component Identifier, as defined in this section, in the parameters parameter of the ExecutableInterface::execute operation. SCA752 When the create operation deploys an ApplicationComponent via an ExecutableDeviceComponent, it shall include a ComponentRegistry IOR, as defined in this section, in the parameters parameter of the ExecutableInterface::execute operation when the SAD componentinstantiation stringifiedobjectref element is null value. SCA77 When the create operation deploys an ApplicationComponent via an ApplicationComponentFactoryComponent, it shall provide the Component Identifier parameter as defined in this section.

The Component Identifier is a CF::Properties type with an id element set to "COMPONENT_IDENTIFIER" and a value element set to a string in the format of "Component_Instantiation_Identifier:Application_Name". The Component_Instantiation_Identifier is the componentinstantiation element id attribute for the component in the application's SAD file. The Application_Name field is identical to the create operation's input name parameter. The Application_Name field provides a specific instance qualifier for executed components. The ComponentRegistry IOR is a CF::Properties type with an id element set to "COMPONENT_REGISTRY_IOR" and a value of the element set to a stringified ComponentRegistry IOR that the ApplicationComponent should use for registration.

SCA81 The create operation shall pass the values of the execparam properties of the componentinstantiation componentproperties element contained in the SAD, as parameters to the execute operation when an ApplicationComponent is deployed via an ExecutableDeviceComponent. The create operation passes execparam parameters values as string values.
The \textit{create} operation may obtain a component in accordance with the SAD via an ApplicationComponentFactoryComponent. SCA83 The \textit{create} operation, when creating an ApplicationComponent from an ApplicationComponentFactoryComponent, shall pass the \texttt{componentinstantiation componentfactoryref} element properties whose \texttt{kindtype} element is "factoryparam" as the qualifiers parameter to the referenced ApplicationComponentFactoryComponent's \textit{createComponent} operation.

SCA85 The \textit{create} operation shall establish connections for an AssemblyComponent which are specified in the SAD \texttt{connections} element. SCA86 The \textit{create} operation shall use the SAD \texttt{connectinterface} element \texttt{id} attribute as part of the unique identifier for a specific connection when provided. SCA87 The \textit{create} operation shall create a unique identifier and use it to designate a connection when no SAD \texttt{connectinterface} element \texttt{id} attribute is specified. Connection ids are formed if they are not provided by the Domain Profile or if there are multiple instances of the same application type. SCA88 For connections to an event channel, the \textit{create} operation shall connect a \texttt{CosEventComm::PushConsumer} or \texttt{CosEventComm::PushSupplier} object to the event channel as specified in the SAD's \texttt{domainfinder} element. SCA89 The \textit{create} operation shall create the specified event channel if the event channel does not exist.

SCA90 The \textit{create} operation shall configure the ManageableApplicationComponent(s) indicated by the \texttt{assemblycontroller} element in the SAD that have properties with a \texttt{kindtype} of "configure" and a \texttt{mode} of "readwrite" or "writeonly" along with the union of properties contained in the input \texttt{initConfiguration} parameter of the \textit{create} operation.

For connections to a ServiceComponent using the servicename type of the \texttt{domainfinder} element, the \textit{create} operation will search for a matching service from the set of service name identifiers that have been registered with the domain. For connections to a ServiceComponent using the servicetype type of the \texttt{domainfinder} element, the \textit{create} operation will search for a matching type from the set of service types that have been registered with the domain. The search strategy used to select a specific instance of a service type when multiple instances of the same service type have been registered with the domain is implementation dependent.

SCA98 For \texttt{domainfinder} element "servicetype" connections to a ServiceComponent whose service type is provided by a service contained within a \texttt{channel} element servicelist, the \textit{create} operation shall only attempt to establish connections to services within the list. If multiple instances of the same service type exist within the servicelist, then an implementation dependent search strategy is used to select a specific instance.

SCA99 The \textit{create} operation shall, upon successful application creation, write an ADMINISTRATIVE\_EVENT log record.

SCA100 The \textit{create} operation shall, upon unsuccessful application creation, write a FAILURE\_ALARM log record.

SCA101 The \textit{create} operation shall send a ComponentChangeEvent\_Type event to the Outgoing Domain Management event channel upon successful creation of an application. For this event:

1. The \texttt{producerId} is the identifier attribute of the ApplicationFactoryComponent;
2. The \texttt{componentChange} is ADDED;
3. The \texttt{domainComponent} is the returned ApplicationManagerComponent's \texttt{CF::ComponentType}.
The following steps demonstrate one scenario of the behavior of an application factory for the creation of an application:

1. Client invokes the \textit{create} operation. Evaluate the Domain Profile for available devices that meet the application's memory and processor requirements, available dependent applications, and dependent libraries needed by the application;
2. Allocate the device(s) memory and processor utilization. Update the memory and processor utilization of the devices;
3. Create an instance of an \texttt{ApplicationManager}, if the requested application can be created;
4. \texttt{ApplicationFactoryComponent} creates a \texttt{ComponentRegistry} instance to be used for deployed application component registration;
5. Load the application software modules on the devices using the appropriate \texttt{Device(s)} interface provided the application software modules haven't already been loaded;
6. Execute the application software modules on the devices using the appropriate \texttt{ExecutableInterface} instance as indicated by the application's software profile;
7. The deployed application components register via the \texttt{ComponentRegistry} interface;
8. The \textit{create} operation writes a log message indicating that a new application was created;
9. Return the \texttt{ApplicationManager} object reference.

Figure 3-29 is a sequence diagram depicting the behavior as described above.
3.1.3.3.2.2.4 Constraints
SCA174 An ApplicationFactoryComponent shall realize the ApplicationFactory interface.

3.1.3.3.2.3 DomainManagerComponent
3.1.3.3.2.3.1 Description
The DomainManagerComponent is used for the control and configuration of the system domain.

Figure 3-29: ApplicationFactory Application Creation Behavior
3.1.3.3.2.3.2 Associations

- **domainEventChannel**: A DomainManagerComponent creates event channels, sends event messages to event channels, disconnects producers and consumers from event channels and may destroy event channels.
- **fileManager**: A DomainManagerComponent creates and manages FileManagerComponents within the domain.
- **appFactories**: A DomainManagerComponent restores any ApplicationFactoryComponent(s) instantiated in previous incarnations of the domain.
- **appManagerComponent**: A DomainManagerComponent contains a set of instantiated applications.
- **domainComponent**: A DomainManagerComponent removes and disconnects, as necessary, components registered within the domain.
- **utilityComponent**: A DomainManagerComponent utilizes the capabilities provided by ServiceComponent(s) within the domain.
- **componentRegistry**: A DomainManagerComponent creates componentRegistries that contain an inventory of the registered components.

3.1.3.3.2.3.3 Semantics

**SCA177** The DomainManagerComponent identifier shall be identical to the *domainmanagerconfiguration* element *id* attribute of the DMD file.

A DomainManagerComponent uses the information in its DMD for determining:

1. Services to be connected to (for example, *log*);
2. Properties for its configuration;
3. Platform Channels for deployment considerations;
4. The DomainManagerComponent's identifier attribute value which is the DMD's id attribute value.

Since a log service is not a required component, a DomainManagerComponent may, or may not have access to a log. However, if log service(s) are available, a DomainManagerComponent may use one or more of them. SCA178 A DomainManagerComponent shall define its utilized ServiceComponents in the DMD.

SCA179 A DomainManagerComponent shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log, when the managers attribute is obtained by a client.

SCA180 A DomainManagerComponent shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log, when the applications attribute is obtained by a client.

SCA181 A DomainManagerComponent shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log, when the applicationFactories attribute is obtained by a client.

SCA182 A DomainManagerComponent shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log, when the fileMgr attribute is obtained by a client.
A DomainManagerComponent may begin to use a service specified in the DMD only after the service has successfully registered with the DomainManagerComponent via the ComponentRegistry::registerComponent operation.

SCA184 A DomainManagerComponent shall create its own FileManagerComponent that consists of all registered DeviceManagerComponent's FileSystemComponents.

SCA185 Upon system startup, a DomainManagerComponent shall restore ApplicationFactoryComponents for AssemblyComponents that were previously installed by the DomainManager::installApplication operation.

SCA186 A DomainManagerComponent shall add the restored application factories to the DomainManager interface applicationFactories attribute.

SCA187 A DomainManagerComponent shall create the Incoming Domain Management and Outgoing Domain Management event channels.

SCA189 The registerComponent operation shall write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent log upon successful component registration.

SCA191 The registerComponent operation shall write a FAILURE_ALARM log record to a DomainManagerComponent log upon unsuccessful component registration.

SCA193 The registerComponent operation shall send a ComponentChangeEvent EventType event to the Outgoing Domain Management event channel, upon successful registration of a component. For this event:
1. The producerId is the identifier attribute of the DomainManagerComponent;
2. The componentChange is ADDED;
3. The domainComponent is the input registering component's CF::ComponentType.
SCA194 The `registerComponent` operation shall establish any pending connections from the registeringComponent.

SCA144 The `registerComponent` operation shall register all of the components identified within the registeringComponent's specializedInfo field when the registeringComponent's type field is `DEVICE_MANAGER_COMPONENT`.

The registerComponent operation upon successful BasePlatformComponent registration adds the registeringComponent to its corresponding registered DeviceManagerComponent's CF::ManagerInfo deployedComponents in the DomainManagerComponent's managers attribute.

SCA149 The `unregisterComponent` operation shall unregister all of the associated (i.e. registered) components from the component being unregistered when its type field is a `DEVICE_MANAGER_COMPONENT`.

The `unregisterComponent` operation may destroy the Event Service event channel when no more consumers and producers are connected to it.

SCA195 The `unregisterComponent` operation shall, upon the successful unregistration of a component, write an ADMINISTRATIVE_EVENT log record to a DomainManagerComponent's log.

SCA196 The `unregisterComponent` operation shall send a ComponentChangeEventType event to the Outgoing Domain Management event channel, upon successful unregistration of a component. For this event:

1. The `producerId` is the identifier attribute of the DomainManagerComponent;
2. The `componentChange` is REMOVED;
3. The `domainComponent` is the unregistered component's CF::ComponentType.

SCA197 The `unregisterComponent` operation shall, upon unsuccessful unregistration of a component, write a FAILURE_ALARM log record to a DomainManagerComponent's log.

SCA198 The `unregisterComponent` operation shall disconnect any connections (including those made to the Event Service event channels) to the unregistering component indicated by the input identifier parameter. SCA199 Connections broken as a result of this `unregisterComponent` operation shall be considered as pending for future connections when the component to which the component was connected still exists.

SCA201 The `registerComponent` operation shall establish any connections for the DeviceManagerComponent indicated by the input registeringComponent parameter, which are specified in the connections element of the DeviceManagerComponent's DCD file, that are possible with the current set of registered components. Connections not currently possible are left unconnected pending future component registrations.

SCA202 For connections established for an Event Service's event channel, the `registerComponent` operation shall connect a `CosEventComm::PushConsumer` or `CosEventComm::PushSupplier` object to the event channel as specified in the DCD's `domainfinder` element. SCA203 If the event channel does not exist, the `registerComponent` operation shall create the event channel.

SCA204 The `registerComponent` operation shall mount the DeviceManagerComponent's FileSystemComponent to the DomainManagerComponent's
FileManagerComponent. SCA205 The mounted \textit{FileSystem} name shall have the format, "/DomainName/HostName", where DomainName is the name of the domain and HostName is the identifier of the input registeringManager.

The DomainManagerComponent associates the input DeviceManagerComponent's registered components with the DeviceManagerComponent in order to support the unregisterComponent operation for a DeviceManagerComponent.

SCA208 The \textit{unregisterComponent} operation shall disconnect the established connections (including those made to the Event Service event channels) of the un registering component as well as for its registered components that have not already been disconnected by the unregistering component when the unregisteringComponent's type field is DEVICE_MANAGER_COMPONENT.

The \textit{unregisterComponent} operation may destroy the Event Service channel when no more consumers and producers are connected to it.

SCA210 The \textit{unregisterComponent} operation shall unmount all DeviceManagerComponent's file systems from its FileManagerComponent when the unregistering component's type field is DEVICE_MANAGER_COMPONENT.

3.1.3.3.2.3.4 Constraints

SCA214 A DomainManagerComponent shall realize the DomainManager interface.

SCA532 A DomainManagerComponent shall fulfill the BaseComponent requirements. SCA559 A DomainManagerComponent shall realize the DomainInstallation interface. SCA560 A DomainManagerComponent shall realize EventChannelRegistry interface.

3.1.3.3.2.4 DeviceManagerComponent

3.1.3.3.2.4.1 Description

A DeviceManagerComponent manages a set of DeviceComponent and ServiceComponent components on a node. The DeviceManagerComponent provides the capability of starting up the managed component(s)' main processes on a given node.
3.1.3.3.2.4.2 Associations

- nodeFileSystem: A DeviceManagerComponent creates FileSystemComponent(s) and mounts them on a FileManagerComponent, if applicable.
- deployedCF: A DeviceManagerComponent deploys, initializes and configures PlatformComponentFactoryComponent(s) as necessary.
- deployedComponent: A DeviceManagerComponent deploys, initializes and configures BasePlatformComponent(s) as necessary.
- componentRegistry: A DeviceManagerComponent creates componentRegistries that contain an inventory of the created BasePlatformComponent(s).
- domainFullRegistrar: A DeviceManagerComponent unregisters componentRegistries from a DomainManagerComponent via its associated FullManagerRegistry instance.
- domainRegistrar: A DeviceManagerComponent registers with a DomainManagerComponent via the DomainManagerComponent's associated ComponentRegistry instance.

3.1.3.3.2.4.3 Semantics

SCA215 A DeviceManagerComponent shall be accompanied by the appropriate Domain Profile files per section 3.1.3.6.
SCA216 A DeviceManagerComponent upon start up shall register with a DomainManagerComponent via the ComponentRegistry interface. The CF::ComponentType's type field of the registering DeviceManagerComponent is DEVICE_MANAGER_COMPONENT. SCA577 The registering DeviceManagerComponent CF::ComponentType's specializedInfo field shall contain a CF::ManagerInfo structure with an id...
of MANAGER_INFO_ID and a value of type CF::ManagerInfo that holds the
BasePlatformComponents that have been deployed by or registered with a
DeviceManagerComponent.

SCA572 The DeviceManagerComponent shall assign a CF::ComponentType's specializedInfo allocation properties with an id of ALLOCATION_PROPS_ID and a value of type CF::AllocationProperties when no CF::AllocationProperties are supplied by the deployed BasePlatformComponent.

A DeviceManagerComponent uses the information in its DCD for determining:

1. Services to be deployed for this DeviceManagerComponent (for example, log(s));
2. DeviceComponents to be created for this DeviceManagerComponent (when the DCD deployondevice element is not specified then the DCD componentinstantiation element is deployed on the same hardware device as the device manager);
3. DeviceComponents to be deployed on (executing on) another DeviceComponent;
4. DeviceComponents to be aggregated to another DeviceComponent;
5. Mount point names for file systems;
6. The DeviceManagerComponent's identifier attribute value which is the DCD's id attribute value;
7. DomainManagerComponent's ComponentRegistry references;
8. Values for its properties.

SCA217 A DeviceManagerComponent shall create FileSystemComponents implementing the FileSystem interface for each OS file system. SCA218 If multiple FileSystemComponents are to be created, the DeviceManagerComponent shall mount created FileSystemComponents to a FileManagerComponent (widened to a FileSystemComponent through the CF::ManagerInfo's FileSys field). The mount points used for the created file systems are identical to the values identified in the filesystemnames element of the DeviceManagerComponent's DCD.

The DeviceManagerComponent can deploy DeviceComponents, PlatformComponentFactoryComponents and ServiceComponents directly (e.g. thread, posix_spawn) by using an ExecutableDeviceComponent or by PlatformComponentFactoryComponent. These components may register with the DeviceManagerComponent via the ComponentRegistry::registerComponent operation. Upon successful BasePlatformComponent deployment, the DeviceManagerComponent adds the deployed component to its deployedComponents attribute.

Successful BasePlatformComponent registration via the registerComponent operation adds the registeringComponent to the DeviceManagerComponent's CF::ManagerInfo deployedComponents.

SCA442 When a DeviceComponent is deployed by the DeviceManagerComponent, the DeviceManagerComponent shall supply execute operation parameters for a device consisting of:
1. Component Registry IOR when the DCD componentinstantiation stringifiedobjectref element is null value - The ID is "COMPONENT_REGISTRY_IOR" and the value is a string that is the ComponentRegistry stringified IOR;
2. Device Identifier - The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD componentinstantiation id attribute;
3. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an AggregateDeviceComponent stringified IOR (this parameter is only used when the DCD componentinstantiation element represents the child device of another componentinstantiation element);
4. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element (a DeviceManagerComponent passes execparam parameters' IDs and values as string values).

SCA224 A DeviceManagerComponent shall use the stacksize and priority elements as specified in the componentinstantiation element's SPD implementation code for the execute operation options parameter.

SCA449 If a PlatformComponentFactoryComponent is deployed by the DeviceManagerComponent, a DeviceManagerComponent shall supply execute operation parameters consisting of:
1. Component Registry IOR - The ID is "COMPONENT_REGISTRY_IOR" and the value is a string that is the ComponentRegistry stringified IOR when the DCD componentinstantiation stringifiedobjectref element is null value;
2. Component Identifier - The ID is "COMPONENT_IDENTIFIER" and the value is a string that corresponds to the DCD componentinstantiation id attribute;
3. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element (a DeviceManagerComponent passes execparam parameters' IDs and values as string values).

SCA538 If a ServiceComponent is deployed by the DeviceManagerComponent, a DeviceManagerComponent shall supply execute operation parameters consisting of:
1. Component Registry IOR - The ID is "COMPONENT_REGISTRY_IOR" and the value is a string that is the ComponentRegistry stringified IOR when the DCD componentinstantiation stringifiedobjectref element is null value;
2. Service Name when the DCD componentinstantiation usagename element is non-null value - The ID is "SERVICE_NAME" and the value is a string in an "identifier\type" format that corresponds to the DCD componentinstantiation usagename element;
3. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element (a DeviceManagerComponent passes execparam parameters' IDs and values as string values).

SCA438 When a DeviceComponent is deployed via PlatformComponentFactoryComponent, the DeviceManagerComponent shall supply the following properties as the qualifiers parameter to the referenced ComponentFactory::createComponent operation:
1. Device Identifier - The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD componentinstantiation id attribute;

2. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an AggregateDeviceComponent stringified IOR (this parameter is only used when the DCD componentinstantiation element represents the child device of another componentinstantiation element);

3. The componentinstantiation componentfactoryref element properties whose kindtype element is "factoryparam".

SCA226 The DeviceManagerComponent shall use the stacksize and priority elements as specified in the componentinstantiation element's SPD implementation code as qualifiers parameter for the ComponentFactory::createComponent operation.

SCA439 When a ServiceComponent is deployed via a PlatformComponentFactoryComponent, the DeviceManagerComponent shall supply the following properties as the qualifiers parameter to the referenced PlatformComponentFactoryComponent's createComponent operation:

1. Service Name when the DCD componentinstantiation usagename element is non-null value - The ID is "SERVICE_NAME" and the value is a string in an "identifierType" format that corresponds to the DCD componentinstantiation usagename element;

2. The componentinstantiation componentfactoryref element properties whose kindtype element is "factoryparam".

SCA227 The DeviceManagerComponent shall initialize deployed components that are instantiated by the DeviceManagerComponent provided they realize the LifeCycle interface.

SCA228 After component initialization, the DeviceManagerComponent shall configure deployed components that are instantiated by the DeviceManagerComponent, provided they realize the PropertySet interface. SCA229 The DeviceManagerComponent shall configure a DCD's componentinstantiation element provided the componentinstantiation element has configure readwrite or writeonly properties with values.

SCA230 The DeviceManagerComponent shall register a registering component with the DomainManagerComponent when the DeviceManagerComponent has previously registered with the DomainManagerComponent. These BasePlatformComponents are registered with the DomainManagerComponent after they are registered with the DeviceManagerComponent.

The registeringComponent's CF::ComponentType specializedInfo contains an id of DEVICE_MANAGER_ID and a value of DeviceManagerComponent's identifier attribute.

SCA231 The registerComponent operation shall, upon unsuccessful component registration, write a FAILURE_ALARM log record to a domain manager's log.

SCA573 The DeviceManagerComponent shall unregister the PlatformComponentFactoryComponent's BasePlatformComponents when a PlatformComponentFactoryComponent unregisters with the DeviceManagerComponent.

SCA232 The unregisterComponent operation shall, upon unsuccessful unregistration of a component, write a FAILURE_ALARM log record to a DomainManagerComponent's log.

SCA233 The unregisterComponent operation shall unregister the registered component specified by the input identifier parameter from the DomainManagerComponent if it is registered with the
DeviceManagerComponent and the DeviceManagerComponent is not shutting down. An unregistering component is removed from DeviceManagerComponent's deployedComponents attribute.

Figure 3-32: depicts a device manager startup scenario as follows:

1. Process DCD and create a ComponentRegistry instance;
2. Deploy platform components passing the component registry object reference for registration;
3. Deployed components register with the ComponentRegistry instance;
4. Initialize all deployed components;
5. Configure all deployed components.

![Device Manager Startup Scenario](image-url)

**Figure 3-32: Device Manager Startup Scenario**
3.1.3.3.2.4.4 Constraints
SCA234 A DeviceManagerComponent shall realize the ComponentIdentifier interface. SCA235 A DeviceManagerComponent shall fulfill the BaseComponent requirements. SCA236 Each mounted file system name shall be unique within a DeviceManagerComponent. SCA561 A DeviceManagerComponent shall realize the DeploymentAttributes interface. SCA562 A DeviceManagerComponent shall realize the ReleasableManager interface.

3.1.3.4 Base Device

3.1.3.4.1 Interfaces
The device interfaces are for the implementation and management of logical devices within the domain. The devices within the domain may be simple devices with no loadable, executable, or aggregate device behavior, or devices with a combination of these behaviors. The device interfaces are AggregateDevice, AggregateDeviceAttributes, CapacityManagement, DeviceAttributes, LoadableInterface AdministratableInterface and ExecutableInterface. Base Device Interfaces are implemented using interface definitions expressed in a Platform Specific representation of one of the Appendix E enabling technologies.

3.1.3.4.1.1 AdministratableInterface
3.1.3.4.1.1.1 Description
The AdministratableInterface interface defines an administrative attribute for any logical device in the domain. A logical device provides the adminState attribute, which describes the administrative state of the device.

3.1.3.4.1.1.2 UML
The AdministratableInterface interface UML is depicted in Figure 3-33.

```
«interface»
CF::AdministratableInterface
+ adminState: AdminType
```

Figure 3-33: AdministratableInterface Interface UML

3.1.3.4.1.3 Types
3.1.3.4.1.3.1 AdminType
This is an IDL enumeration type that defines a device's administrative states. The administrative state indicates the permission to use or prohibition against using the device.

```c
enum AdminType
{
    LOCKED,
    SHUTTING_DOWN,
    UNLOCKED
};
```
3.1.3.4.1.4 Attributes

3.1.3.4.1.4.1 adminState

SCA243 The adminState attribute shall return the device's admin state value.

SCA244 The adminState attribute shall only allow the setting of LOCKED and UNLOCKED values, where setting LOCKED is only effective when the adminState attribute value is UNLOCKED, and setting UNLOCKED is only effective when the adminState attribute value is LOCKED or SHUTTING_DOWN. Illegal state transition commands are ignored.

attribute AdminType adminState;

3.1.3.4.1.5 Operations

N/A.

3.1.3.4.1.2 CapacityManagement

3.1.3.4.1.2.1 Description

The CapacityManagement interface defines additional capabilities and an attribute for any logical device in the domain. A logical device provides the following attribute and operations:

1. Usage State Management Attribute - This information describes the usage states of the device;

2. Capacity Operations - In order to use a device, certain capacities (e.g., memory, performance, etc.) are obtained from the device. A device may have multiple capacities which need to be allocated, since each device has its own unique capacity model which is described in the associated SPD.

3.1.3.4.1.2.2 UML

The CapacityManagement interface UML is depicted in Figure 3-34.

```
«interface»
CF::CapacityManagement
+ usageState: UsageType
+ allocateCapacity(Properties): boolean
+ deallocateCapacity(Properties): void
```

**Figure 3-34: CapacityManagement Interface UML**

3.1.3.4.1.2.3 Types

3.1.3.4.1.2.3.1 InvalidCapacity

The InvalidCapacity exception returns the capacities that are not valid for this device.

```
exception InvalidCapacity {string msg; Properties capacities;};
```

3.1.3.4.1.2.3.2 UsageType

This is an IDL enumeration type that defines the device's usage states. The usage state indicates which of the following states a device is in:

- IDLE - not in use,
ACTIVE - in use, with capacity remaining for allocation, or
BUSY - in use, with no capacity remaining for allocation.

```c
enum UsageType
{
    IDLE,
    ACTIVE,
    BUSY
};
```

### 3.1.3.4.1.2.4 Attributes

#### 3.1.3.4.1.2.4.1 usageState.

SCA248 The readonly usageState attribute shall return the device's usage state (IDLE, ACTIVE, or BUSY). UsageState indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for allocation at that instant.

```c
readonly attribute UsageType usageState;
```

### 3.1.3.4.1.2.5 Operations

#### 3.1.3.4.1.2.5.1 allocateCapacity

**Brief Rationale**

The `allocateCapacity` operation provides the mechanism to request and allocate capacity from the device.

**Synopsis**

```c
boolean allocateCapacity (in Properties capacities) raises (InvalidCapacity, InvalidState);
```

**Behavior**

SCA250 The `allocateCapacity` operation shall reduce the current capacities of the device based upon the input capacities parameter, when usageState attribute is not BUSY.

SCA251 The `allocateCapacity` operation shall set the device's usageState attribute to BUSY, when the device determines that it is not possible to allocate any further capacity. SCA252 The `allocateCapacity` operation shall set the usageState attribute to ACTIVE, when capacity is being used and any capacity is still available for allocation.

SCA253 The `allocateCapacity` operation shall only accept properties for the input capacities parameter which are simple properties whose `kindtype` is "allocation" and whose `action` element is "external" contained in the component's SPD.

**Returns**

SCA254 The `allocateCapacity` operation shall return TRUE, if the capacities have been allocated, or FALSE, if not allocated.

**Exceptions/Errors**

SCA255 The `allocateCapacity` operation shall raise the InvalidCapacity exception, when the input capacities parameter contains invalid properties or when attributes of those CF::Properties contain an unknown `id` or a value of the wrong data type.
3.1.3.4.1.2.5.2.1 Brief Rationale

The `deallocateCapacity` operation provides the mechanism to return capacities back to the device, making them available to other users.

3.1.3.4.1.2.5.2.2 Synopsis

```c
void deallocateCapacity (in Properties capacities) raises (InvalidCapacity, InvalidState);
```

3.1.3.4.1.2.5.2.3 Behavior

**SCA257** The `deallocateCapacity` operation shall increment the current capacities of the device based upon the input capacities parameter.

**SCA258** The `deallocateCapacity` operation shall set the usageState attribute to ACTIVE when, after adjusting capacities, any of the device's capacities are still being used.

**SCA259** The `deallocateCapacity` operation shall set the usageState attribute to IDLE when, after adjusting capacities, none of the device's capacities are still being used.

3.1.3.4.1.2.5.2.4 Returns

This operation does not return any value.

3.1.3.4.1.2.5.2.5 Exceptions/Errors

![State Transition Diagram for allocateCapacity and deallocateCapacity](image-url)
SCA261 The *deallocateCapacity* operation shall raise the InvalidCapacity exception, when the capacity ID is invalid or the capacity value is the wrong type. The InvalidCapacity exception msg parameter describes the reason for the exception.

3.1.3.4.1.3 DeviceAttributes

3.1.3.4.1.3.1 Description

The *DeviceAttributes* interface inherits the *ComponentIdentifier* interface.

The *DeviceAttributes* interface defines attributes for any logical device in the domain. A logical device may provide the following attributes:

1. Operational State Management - This information describes the operational states of the device.

3.1.3.4.1.3.2 UML

The *DeviceAttributes* interface UML is depicted in Figure 3-36.

![Diagram of DeviceAttributes Interface UML](image)

**Figure 3-36: DeviceAttributes Interface UML**

3.1.3.4.1.3.3 Types

3.1.3.4.1.3.3.1 OperationalType

This is an IDL enumeration type that defines a device’s operational states. The operational state indicates whether or not the object is functioning.

```cpp
enum OperationalType {
    ENABLED,
    DISABLED
};
```

3.1.3.4.1.3.4 Attributes

3.1.3.4.1.3.4.1 operationalState

SCA263 The readonly operationalState attribute shall return the device’s operational state (ENABLED or DISABLED). The operational state indicates whether or not the device is active.

```cpp
readonly attribute OperationalType operationalState;
```
3.1.3.4.1.3.5 Operations
N/A.

3.1.3.4.1.4 AggregateDeviceAttributes
3.1.3.4.1.4.1 Description
The \textit{AggregateDeviceAttributes} interface defines attributes for any logical device in the domain. A logical device may provide the \texttt{compositeDevice} attribute which contains a reference to a component that aggregates a sequence of child devices.

3.1.3.4.1.4.2 UML
The \textit{AggregateDeviceAttributes} interface UML is depicted in Figure 3-37.

![AggregateDeviceAttributes Interface UML](image)

**Figure 3-37: AggregateDeviceAttributes Interface UML**

3.1.3.4.1.4.3 Types
N/A.

3.1.3.4.1.4.4 Attributes
3.1.3.4.1.4.4.1 compositeDevice
SCA266 The readonly \texttt{compositeDevice} attribute shall return the object reference of the \texttt{AggregateDeviceComponent}. SCA267 The readonly \texttt{compositeDevice} attribute shall return a nil object reference when this \texttt{DeviceComponent} is not a parent.

```c
readonly attribute AggregateDevice compositeDevice;
```

3.1.3.4.1.4.5 Operations
N/A.

3.1.3.4.1.5 LoadableInterface
3.1.3.4.1.5.1 Description
This interface provides software loading and unloading behaviors to a device.

3.1.3.4.1.5.2 UML
The \textit{LoadableInterface} interface UML is depicted in Figure 3-38.

![LoadableInterface Interface UML](image)
3.1.3.4.1.5.3 Types
3.1.3.4.1.5.3.1 LoadType
The LoadType defines the type of load to be performed. The load types are in accordance with the code element within the softpkg element's implementation element, which is defined in Appendix D-1.

```cpp
enum LoadType {
    KERNEL_MODULE,
    DRIVER,
    SHARED_LIBRARY,
    EXECUTABLE
};
```

3.1.3.4.1.5.3.2 InvalidLoadKind
The InvalidLoadKind exception indicates that the loadable device is unable to load the type of file designated by the loadKind parameter.

```cpp
exception InvalidLoadKind{};
```

3.1.3.4.1.5.3.3 LoadFail
The LoadFail exception indicates that the load operation failed due to device dependent reasons. The LoadFail exception indicates that an error occurred during an attempt to load the input file onto the loadable device. The error number indicates a CF::ErrorNumberType. The message is component-dependent, providing additional information describing the reason for the error.

```cpp
exception LoadFail { ErrorNumberType errorNumber; string msg; }
```

3.1.3.4.1.5.4 Attributes
N/A.

3.1.3.4.1.5.5 Operations
3.1.3.4.1.5.5.1 load
3.1.3.4.1.5.5.1.1 Brief Rationale
The load operation provides the mechanism for loading software on a loadable device. The loaded software may be subsequently executed on the device, if the device is an executable device.

3.1.3.4.1.5.5.1.2 Synopsis
```cpp
void load (in FileSystem fs, in string fileName, in LoadType loadKind) raises (InvalidState, InvalidLoadKind, InvalidFileName, LoadFail);
```

3.1.3.4.1.5.5.1.3 Behavior
SCA268 The load operation shall load the file identified by the input fileName parameter on the DeviceComponent based upon the input loadKind parameter. The input fileName parameter is a pathname relative to the file system identified by the input fs parameter.
SCA269 Multiple *loads* of the same file as indicated by the input `fileName` parameter shall not result in an exception. However, the `load` operation should account for this multiple load so that the `unload` operation behavior can be performed.

3.1.3.4.1.5.5.1.4 Returns
This operation does not return any value.

3.1.3.4.1.5.5.1.5 Exceptions/Errors

SCA271 The `load` operation shall raise the `InvalidLoadKind` exception when the input `loadKind` parameter is not supported.

SCA272 The `load` operation shall raise the CF::InvalidFileName exception when the file designated by the input `fileName` parameter cannot be found.

SCA273 The `load` operation shall raise the LoadFail exception when an attempt to load the device is unsuccessful.

3.1.3.4.1.5.5.2 `unload`

3.1.3.4.1.5.5.2.1 Brief Rationale
The `unload` operation provides the mechanism to unload software that is currently loaded.

3.1.3.4.1.5.5.2.2 Synopsis

```java
void unload (in string fileName) raises (InvalidState, InvalidFileName);
```

3.1.3.4.1.5.5.2.3 Behavior

SCA274 The `unload` operation shall unload the file identified by the input `fileName` parameter from the loadable device when the number of unload requests matches the number of load requests for the indicated file.

3.1.3.4.1.5.5.2.4 Returns
This operation does not return a value.

3.1.3.4.1.5.5.2.5 Exceptions/Errors

SCA276 The `unload` operation shall raise the CF::InvalidFileName exception when the file designated by the input `fileName` parameter cannot be found.

3.1.3.4.1.6 ExecutableInterface

3.1.3.4.1.6.1 Description
This interface provides execute and terminate behavior for a device.

3.1.3.4.1.6.2 UML
The `ExecutableInterface` interface UML is depicted in Figure 3-39.
3.1.3.4.1.6.3 Types

3.1.3.4.1.6.3.1 InvalidProcess

The InvalidProcess exception indicates that a process or thread, as identified by the executionId parameter, does not exist on this device. The errorNumber parameter indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

```plaintext
exception InvalidProcess { ErrorNumberType errorNumber; string msg; }
```

3.1.3.4.1.6.3.2 InvalidFunction

The InvalidFunction exception indicates that a function, as specified by the input entry point options parameter, hasn't been loaded on this device.

```plaintext
exception InvalidFunction{}
```

3.1.3.4.1.6.3.3 ExecutionID_Type

The ExecutionID_Type contains information for a process and thread id within the system. The ExecutionID_Type is unique to the processor operating system that created the process and thread. The threadId field is the thread id provided by the operating system when a thread was created to execute the function specified in the entry point options parameter. The processId field is the process identifier provided by the operating system when a process was created either to execute a file or a function as specified in the entry point options parameter. The processCollocation field, when specified, is the value of the process collocation options parameter. The cores field is the value of the processor cores used to execute the process.

```plaintext
struct ExecutionID_Type
{
    unsigned long long threadId;
    unsigned long long processId;
    string processCollocation;
    CF::ULongSeq cores;
};
```
3.1.3.4.1.6.3.4 InvalidParameters

The InvalidParameters exception indicates the input parameters are invalid on the execute operation. The InvalidParameters exception is raised when there are invalid execute parameters. The invalidParms parameter is a list of invalid parameters specified in the execute operation.

```cpp
exception InvalidParameters { Properties invalidParms; }
```

3.1.3.4.1.6.3.5 InvalidOptions

The InvalidOptions exception indicates the input options are invalid on the execute operation. The invalidOpts parameter is a list of invalid options specified in the execute operation.

```cpp
exception InvalidOptions { Properties invalidOpts; }
```

3.1.3.4.1.6.3.6 STACK_SIZE_ID

STACK_SIZE_ID is the identifier for the execute operation options parameter. STACK_SIZE_ID is used to set the operating system's process/thread stack size. The value of a stack size is an unsigned long.

```cpp
const string STACK_SIZE_ID = "STACK_SIZE";
```

3.1.3.4.1.6.3.7 PRIORITY_ID

PRIORITY_ID is the identifier for the execute operation options parameters. PRIORITY_ID is used to set the operating system's process/thread priority. The value of a priority is an unsigned long.

```cpp
const string PRIORITY_ID = "PRIORITY";
```

3.1.3.4.1.6.3.8 EXEC_DEVICE_PROCESS_SPACE

EXEC_DEVICE_PROCESS_SPACE is the identifier for the execute operation. PROCESS_COLLOCATION_ID option parameter.

```cpp
const string EXEC_DEVICE_PROCESS_SPACE = "DEVICE";
```

3.1.3.4.1.6.3.9 PROCESS_COLLOCATION_ID

PROCESS_COLLOCATION_ID is the identifier for the execute operation options parameter. PROCESS_COLLOCATION_ID is used to select the process from within which the entry point function must be invoked. A PROCESS_COLLOCATION_ID value of EXEC_DEVICE_PROCESS_SPACE means the entry point is invoked from within the process of the Executable Device Component. A PROCESS_COLLOCATION_ID empty value means a new process is created to invoke the entry point. A PROCESS_COLLOCATION_ID of any other value means the entry point is invoked from within a process associated with that logical process value. The value for a process collocation is a string.

```cpp
const string PROCESS_COLLOCATION_ID = "PROCESS_COLLOCATION";
```

3.1.3.4.1.6.3.10 ENTRY_POINT_ID

ENTRY_POINT_ID is the identifier for the execute operation options parameter. ENTRY_POINT_ID is used to identify the name of the entry point function that must be invoked. The value for an entry point is a string.

```cpp
const string ENTRY_POINT_ID = "ENTRY_POINT";
```
3.1.3.4.1.6.3.11 CORE_AFFINITY_ID
CORE_AFFINITY_ID is the identifier for the execute operation options parameter. CORE_AFFINITY_ID is used to identify the processor core upon which to execute a process. The value for a core affinity is a CF::ULongSeq.

    const string CORE_AFFINITY_ID = "CORE_AFFINITY";

3.1.3.4.1.6.3.12 ExecuteFail
The ExecuteFail exception indicates that the execute operation failed due to device dependent reasons. The ExecuteFail exception indicates that an error occurred during an attempt to invoke the operating system "execute/thread" function on the device. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

    exception ExecuteFail { ErrorNumberType errorNumber; string msg; }

3.1.3.4.1.6.4 Attributes
N/A.

3.1.3.4.1.6.5 Operations
3.1.3.4.1.6.5.1 execute
3.1.3.4.1.6.5.1.1 Brief Rationale
The execute operation provides the mechanism for starting up and executing a software process/thread on a device.

3.1.3.4.1.6.5.1.2 Synopsis
ExecutionID_Type execute (in string filename, in Properties options, in Properties parameters) raises (InvalidState, InvalidFunction, InvalidParameters, InvalidOptions, InvalidFileName, ExecuteFail);

3.1.3.4.1.6.5.1.3 Behavior
SCA279 The execute operation shall execute the file identified by the input filename parameter using the input parameters and options parameters.

SCA280 The execute operation shall map the input parameters (id/value string pairs) parameter as an argument to the operating system "execute/thread" function. The argument (e.g. argv) is an array of character pointers to null-terminated strings where the last member is a null pointer and the first element is the input filename parameter. Thereafter the second element is mapped to the input parameters[0] id, the third element is mapped to the input parameters[0] value and so forth until the contents of the input parameters parameter are exhausted.

The execute operation input options parameters are STACK_SIZE_ID, PRIORITY_ID, PROCESS_COLLOCATION_ID, ENTRY_POINT_ID, and CORE_AFFINITY_ID.

3.1.3.4.1.6.5.1.4 Returns
SCA282 The execute operation shall return a unique ExecutionID_Type for the process/thread that it created. The threadId is zero when no ENTRY_POINT_ID is specified.

3.1.3.4.1.6.5.1.5 Exceptions/Errors
SCA284 The *execute* operation shall raise the *InvalidFunction* exception when the function indicated by the input entry point options parameter is not executed because it does not exist on the device.

SCA285 The *execute* operation shall raise the *CF::InvalidFileName* exception when the file name indicated by the input filename parameter does not exist for the device to be executed.

SCA286 The *execute* operation shall raise the *InvalidParameters* exception when the input parameter ID or value attributes are not valid strings.

SCA287 The *execute* operation shall raise the *InvalidOptions* exception when the input options parameter does not comply with sections 3.1.3.4.1.6.3.6 *STACK_SIZE_ID*, 3.1.3.4.1.6.3.7 *PRIORITY_ID*, 3.1.3.4.1.6.3.9 *PROCESS_COLLOCATION_ID*, 3.1.3.4.1.6.3.10 *ENTRY_POINT_ID*, and 3.1.3.4.1.6.3.11 *CORE_AFFINITY_ID*.

SCA288 The *execute* operation shall raise the *ExecuteFail* exception when the operating system "execute/thread" function is not successful.

3.1.3.4.1.6.5.2 terminate

3.1.3.4.1.6.5.2.1 Brief Rationale
The *terminate* operation provides the mechanism for terminating the execution of a process/thread on a specific device that was started up with the *execute* operation. The *terminate* operation may terminate a process when all threads within that process have been terminated.

3.1.3.4.1.6.5.2.2 Synopsis

```c
void terminate (in ExecutionID_Type executionId) raises (InvalidProcess, InvalidState);
```

3.1.3.4.1.6.5.2.3 Behavior

SCA289 The *terminate* operation shall terminate the execution of the process/thread designated by the executionId input parameter on the device. When the threadId is 0, the specified process will be terminated, including all of the threads it contains. When a specific threadId is provided, only that thread will be terminated.

3.1.3.4.1.6.5.2.4 Returns
This operation does not return a value.

3.1.3.4.1.6.5.2.5 Exceptions/Errors

SCA291 The *terminate* operation shall raise the *InvalidProcess* exception when the executionId does not exist for the device.

3.1.3.4.1.7 AggregateDevice

3.1.3.4.1.7.1 Description
The *AggregateDevice* interface provides the required behavior that is needed to add and remove child devices from a parent device. This interface may be provided via inheritance or as a provides port for any device that is used as a parent device. Child devices use this interface to add or remove themselves to a parent device when being created or torn-down.

3.1.3.4.1.7.2 UML
The *AggregateDevice* interface UML is depicted in Figure 3-40.
3.1.3.4.1.7.3 Types
N/A.
3.1.3.4.1.7.4 Attributes
3.1.3.4.1.7.4.1 devices
SCA292 The readonly devices attribute shall return a list of devices that have been added to this device or a sequence length of zero if the device has no aggregation relationships with other devices.

```plaintext
readonly attribute ObjectSequence devices;
```

3.1.3.4.1.7.5 Operations
3.1.3.4.1.7.5.1 addDevice
3.1.3.4.1.7.5.1.1 Brief Rationale
The `addDevice` operation provides the mechanism to associate a device with another device. When a device changes state or it is being torn down, its associated devices are affected.

```plaintext
void addDevice (in Object associatedDevice, string identifier) raises (InvalidObjectReference);
```

3.1.3.4.1.7.5.1.3 Behavior
SCA293 The `addDevice` operation shall add the input associatedDevice parameter to the AggregateDevice's devices attribute when the associatedDevice associated with the input identifier parameter does not exist in the devices attribute. The associatedDevice is ignored when the identifier duplicated.

3.1.3.4.1.7.5.1.4 Returns
This operation does not return any value.

3.1.3.4.1.7.5.1.5 Exceptions/Errors
SCA295 The `addDevice` operation shall raise the CF::InvalidObjectReference when the input associatedDevice parameter is a nil object reference.

3.1.3.4.1.7.5.2 removeDevice
3.1.3.4.1.7.5.2.1 Brief Rationale
The `removeDevice` operation provides the mechanism to disassociate a device from another device.
3.1.3.4.1.7.5.2.2 Synopsis
void removeDevice (in string identifier) raises (InvalidObjectReference);  

3.1.3.4.1.7.5.2.3 Behavior
SCA296 The removeDevice operation shall remove the device that corresponds to the input identifier parameter from the AggregateDevice's devices attribute.

3.1.3.4.1.7.5.2.4 Returns
This operation does not return any value.

3.1.3.4.1.7.5.2.5 Exceptions/Errors
SCA297 The removeDevice operation shall raise the CF::InvalidObjectReference when the device that corresponds to the input identifier parameter is a nil object reference or does not exist in the AggregateDevice devices attribute.

3.1.3.4.2 Components
Base Device Components provide the structural definitions that will be utilized for the implementation and management of physical devices within the domain. The physical devices within the domain may be simple devices with no loadable, executable, or aggregate device behavior, or devices with a combination of these behaviors.

3.1.3.4.2.1 DeviceComponent
3.1.3.4.2.1.1 Description
A DeviceComponent extends BasePlatformComponent. DeviceComponent contains the core associations and requirements that are used by all of the SCA device components.

Figure 3-41: DeviceComponent UML
3.1.3.4.2.1.2 Associations
   • componentAggregator: A DeviceComponent associates and disassociates itself with another DeviceComponent via an AggregateDeviceComponent.

3.1.3.4.2.1.3 Semantics
A DeviceComponent is a functional abstraction for a set (e.g., zero or more) of physical hardware devices and includes a collection of capability and capacity properties. DeviceComponents communicate with physical hardware devices via device drivers. They are typically used by applications but there is nothing restricting them being utilized by any other type of platform component.

The CF::ComponentType's type field of the registering DeviceComponent is a DEVICE_COMPONENT.

SCA458 A child DeviceComponent shall add itself to a parent device using the executable Composite Device IOR and DEVICE_ID parameters per 3.1.3.3.2.4.3. SCA299 The values associated with the parameters (COMPOSITEDEVICE_IOR, and DEVICE_ID) as described in 3.1.3.3.2.4.3 shall be used to set the DeviceComponent's compositeDevice, and identifier attributes, respectively.

Each DeviceComponent may have a DPD as described in 3.1.3.6. For each DeviceComponent, allocation properties should be defined in its referenced SPD's property file.

3.1.3.4.2.1.3.1 State Model Behavior
SCA237 The releaseObject operation shall assign the LOCKED state to the adminState attribute, when the adminState attribute is UNLOCKED.

SCA238 The releaseObject operation shall call the releaseObject operation on all of the DeviceComponents contained within its referenced AggregateDeviceComponent when the DeviceComponent is a parent device.

SCA239 The releaseObject operation shall cause the removal of a DeviceComponent from the referenced AggregateDeviceComponent of its parent when this DeviceComponent is a child device.

SCA240 The releaseObject operation shall cause the device to be unavailable and released from the operating environment when the adminState attribute transitions to LOCKED. The transition to the LOCKED state signifies that the usageState attribute is IDLE and, if the device is a parent device that its child devices have been removed.

SCA256 The allocateCapacity operation shall raise the CF::InvalidState exception when the DeviceComponent's adminState is not UNLOCKED.

SCA260 The deallocateCapacity operation shall set the adminState attribute to LOCKED as specified in this section.

SCA262 The deallocateCapacity operation shall raise the CF::InvalidState exception, when the DeviceComponent's adminState is LOCKED.

SCA511 The allocateCapacity operation shall raise the CF::InvalidState exception when the DeviceComponent's operationalState is DISABLED.

SCA516 The deallocateCapacity operation shall raise the CF::InvalidState exception, when the DeviceComponent's operationalState is DISABLED.
SCA245 The adminState attribute, upon being commanded to be LOCKED, shall set the adminState to LOCKED for its entire aggregation of DeviceComponents (if it has any). Refer to Figure 3-42 for an illustration of the above state behavior.

The adminState transitions to the LOCKED state when the device's usageState is IDLE and its entire aggregation of DeviceComponents are LOCKED.

SCA247 The DeviceComponent shall send a StateChangeEventTypet event to the Incoming Domain Management event channel, whenever the adminState attribute changes. For this event:

1. The producerId field is the identifier attribute of the DeviceComponent;
2. The sourceId field is the identifier attribute of the DeviceComponent;
3. The stateChangeCategory field is "ADMINISTRATIVE_STATE_EVENT";
4. The stateChangeFrom field is the value of the adminState attribute before the state change;
5. The stateChangeTo field is the value of the adminState attribute after the state change.

![State Transition Diagram for adminState](image)

**Figure 3-42: State Transition Diagram for adminState**

SCA249 The DeviceComponent shall send a StateChangeEventTypet event to the Incoming Domain Management event channel, whenever the usageState attribute changes. For this event:
1. The `producerId` field is the identifier attribute of the DeviceComponent;
2. The `sourceId` field is the identifier attribute of the DeviceComponent;
3. The `stateChangeCategory` field is "USAGE_STATE_EVENT";
4. The `stateChangeFrom` field is the value of the usageState attribute before the state change;
5. The `stateChangeTo` field is the value of the usageState attribute after the state change.

SCA264 The DeviceComponent shall send a StateChangeEventType event to the Incoming Domain Management event channel, whenever the operationalState attribute changes. For this event:
1. The `producerId` field is the identifier attribute of the DeviceComponent;
2. The `sourceId` field is the identifier attribute of the DeviceComponent;
3. The `stateChangeCategory` field is "OPERATIONAL_STATE_EVENT";
4. The `stateChangeFrom` field is the value of the operationalState attribute before the state change;
5. The `stateChangeTo` field is the value of the operationalState attribute after the state change.

The following behavior extends the `LifeCycle::releaseObject` operation requirements (see section 3.1.3.2.1.3.5.2).

SCA241 The `releaseObject` operation shall unregister its device from its DeviceManagerComponent.

Figure 3-43 depicts a release scenario for removal of a child device as follows:
1. Invoke `releaseObject` operation on child DeviceComponent;
2. Remove child DeviceComponent from its parent;
3. Unregister child DeviceComponent from its associated `componentRegistry` instance;
4. Terminate processes / threads associated with the child DeviceComponent.
For this scenario, the child device's adminState = LOCKED. After the device is removed from the OE, its process/threads can be terminated. How a device indicates to its process/thread to terminate is implementation specific.

Figure 3-43: Release Child Device Scenario

Figure 3-44 depicts a release scenario for removal of a parent device as follows:

1. Invoke releaseObject operation on parent DeviceComponent;
2. Obtain list of child DeviceComponents from the parent DeviceComponent's compositeDevice attribute;
3. Remove all of the parent DeviceComponent's child DeviceComponents (see Figure 3-43);
4. Unregister parent DeviceComponent from its associated componentRegistry instance;
5. Terminate processes/threads associated with the parent DeviceComponent.
The releaseObject operation raises the ReleaseError exception when releaseObject is not successful in releasing a logical device due to internal processing errors that occurred within the device being released. See section 3.1.3.2.1.3.5.2.5 for exception handling.

3.1.3.4.2.1.4 Constraints

SCA526 A DeviceComponent shall fulfill the BasePlatformComponent requirements.
SCA534 A DeviceComponent shall realize the DeviceAttributes interface.
SCA535 A DeviceComponent shall realize the AdministratableInterface interface.
SCA536 A DeviceComponent shall realize the CapacityManagement interface.
SCA539 A DeviceComponent shall realize the AggregateDeviceAttributes interface.
SCA563 A DeviceComponent shall realize the LifeCycle interface.

3.1.3.4.2.2 LoadableDeviceComponent

3.1.3.4.2.2.1 Description
The LoadableDeviceComponent extends the DeviceComponent by adding software loading and unloading behavior.
3.1.3.4.2.2.2 Associations

- FileSystem: A LoadableDeviceComponent accesses FileSystemComponent(s) in order to retrieve files which are to be loaded.
- loadedFile: A LoadableDeviceComponent loads and unloads files into the domain.

3.1.3.4.2.2.3 Semantics

SCA306 The *load* operation shall support the load types as stated in the LoadableDeviceComponent's profile supported_*load_types* allocation property. SCA307 When a LoadType is not defined for the LoadableDeviceComponent, the *load* operation shall support all SPD *code* element types. The type field of the registering component is a LOADABLE_DEVICE_COMPONENT.

3.1.3.4.2.2.3.1 State Model Behavior

See section 3.1.3.4.2.1.3.1.

SCA512 The *load* operation shall raise the CF::InvalidState exception if upon entry the LoadableDeviceComponent's operationalState attribute is DISABLED.

SCA513 The *unload* operation shall raise the CF::InvalidState exception if upon entry the LoadableDeviceComponent's operationalState attribute is DISABLED.

SCA270 The *load* operation shall raise the CF::InvalidState exception if upon entry the LoadableDeviceComponent's adminState attribute is either LOCKED or SHUTTING_DOWN.

SCA275 The *unload* operation shall raise the CF::InvalidState exception if upon entry the LoadableDeviceComponent's adminState attribute is LOCKED.

3.1.3.4.2.2.4 Constraints

SCA308 A LoadableDeviceComponent shall realize the *LoadableInterface* interface.

SCA309 A LoadableDeviceComponent shall fulfill the DeviceComponent requirements.

3.1.3.4.2.3 ExecutableDeviceComponent

3.1.3.4.2.3.1 Description

The ExecutableDeviceComponent extends the DeviceComponent by adding execute and terminate process/thread behavior. An ExecutableDeviceComponent accepts executable parameters as specified in the (*ExecutableInterface::execute*) section.
3.1.3.4.2.3.2 Associations

- executableArtifact: An ExecutableDeviceComponent executes and terminates artifact(s) (i.e. processes, executables or modules) within a processing environment.
- fileSystem: An ExecutableDeviceComponent accesses FileSystemComponent(s) in order to retrieve files which are to be executed.

3.1.3.4.2.3.3 Semantics

SCA310 An ExecutableDeviceComponent shall accept the executable parameters as specified in section 3.1.3.4.1.6.5.1.3 (ExecutableInterface::execute). The type field of the registering component is an EXECUTABLE_DEVICE_COMPONENT.

See section 3.1.3.4.2.2.3.

3.1.3.4.2.3.4 Constraints

SCA311 An ExecutableDeviceComponent shall realize the ExecutableInterface interface.
SCA564 An ExecutableDeviceComponent shall realize the LoadableInterface interface.
SCA312 An ExecutableDeviceComponent shall fulfill the DeviceComponent requirements.

3.1.3.4.2.4 AggregateDeviceComponent

3.1.3.4.2.4.1 Description
An AggregateDeviceComponent provides behavior to add and remove child DeviceComponents from a parent DeviceComponent. Child DeviceComponents are provided with and use a reference to an AggregateDeviceComponent to introduce or remove an association between themselves and a parent DeviceComponent that manages the composition. When a parent DeviceComponent changes state or is released, its associated DeviceComponents change correspondingly.

Figure 3-47: AggregateDeviceComponent UML

3.1.3.4.2.4.2 Associations
- aggregatedElements: An AggregateDeviceComponent manages, adds and deletes, DeviceComponent(s) that are children of its associated DeviceComponent.

3.1.3.4.2.4.3 Semantics
N/A.

3.1.3.4.2.4.4 Constraints
SCA313 An AggregateDeviceComponent shall realize the AggregateDevice interface.

3.1.3.5 Framework Services

3.1.3.5.1 Interfaces
Framework Services Interfaces are implemented using interface definitions expressed in a Platform Specific representation of one of the Appendix E enabling technologies.

3.1.3.5.1.1 File
3.1.3.5.1.1.1 Description
The File interface provides the ability to read and write files residing within a compliant, distributed file system. The File interface is modeled after the POSIX/C file interface.
3.1.3.5.1.1.2 UML

Figure 3-48: File Interface UML

3.1.3.5.1.1.3 Types
3.1.3.5.1.1.3.1 IOException
The IOException exception indicates an error occurred during a read or write operation to a file. The error number indicates a CF::ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.
exception IOException { ErrorNumberType errorNumber; string msg; }
3.1.3.5.1.1.3.2 InvalidFilePointer
The InvalidFilePointer exception indicates the file pointer is out of range based upon the current file size.
exception InvalidFilePointer{};

3.1.3.5.1.1.4 Attributes
3.1.3.5.1.1.4.1 fileName
SCA320 The readonly fileName attribute shall return the pathname used as the input fileName parameter of the FileSystem::create operation when the file was created.
readonly attribute string fileName;
3.1.3.5.1.1.4.2 filePointer
SCA321 The readonly filePointer attribute shall return the current file position. The filePointer attribute value dictates where the next read or write will occur.
readonly attribute unsigned long filePointer;

3.1.3.5.1.1.5 Operations
3.1.3.5.1.1.5.1 read
3.1.3.5.1.1.5.1.1 Brief Rationale
Applications require the read operation in order to retrieve data from remote files.
3.1.3.5.1.1.5.1.2 Synopsis
void read (out OctetSequence data, in unsigned long length) raises (IOException);

3.1.3.5.1.1.5.1.3 Behavior

SCA322 The read operation shall read, from the referenced file, the number of octets specified by the input length parameter and advance the value of the filePointer attribute by the number of octets actually read.  SCA323 The read operation shall read less than the number of octets specified in the input length parameter, when an end-of-file is encountered.

3.1.3.5.1.1.5.1.4 Returns

SCA324 The read operation shall return a CF::OctetSequence that equals the number of octets actually read from the file via the out data parameter.  SCA325 If the filePointer attribute value reflects the end of the file, the read operation shall return a zero-length CF::OctetSequence.

3.1.3.5.1.1.5.1.5 Exceptions/Errors

SCA326 The read operation shall raise the IOException when a read error occurs.

3.1.3.5.1.1.5.2 write

3.1.3.5.1.1.5.2.1 Brief Rationale

Applications require the write operation in order to write data to remote files.

3.1.3.5.1.1.5.2.2 Synopsis

void write (in OctetSequence data) raises (IOException);

3.1.3.5.1.1.5.2.3 Behavior

SCA327 The write operation shall write data to the file referenced.  SCA328 The write operation shall increment the filePointer attribute to reflect the number of octets written, when the operation is successful.  SCA329 If the write operation is unsuccessful, the value of the filePointer attribute shall maintain or be restored to its value prior to the write operation call.  If the file was opened using the FileSystem::open operation with an input read_Only parameter value of TRUE, writes to the file are considered to be in error.

3.1.3.5.1.1.5.2.4 Returns

This operation does not return any value.

3.1.3.5.1.1.5.2.5 Exceptions/Errors

SCA330 The write operation shall raise the IOException when a write error occurs.

3.1.3.5.1.1.5.3 sizeOf

3.1.3.5.1.1.5.3.1 Brief Rationale

An application may need to know the size of a file in order to determine memory allocation requirements.

3.1.3.5.1.1.5.3.2 Synopsis

unsigned long sizeOf() raises (FileException);

3.1.3.5.1.1.5.3.3 Behavior

There is no significant behavior beyond the behavior described by the following section.

3.1.3.5.1.1.5.3.4 Returns
SCA331 The $sizeOf$ operation shall return the number of octets stored in the file.

3.1.3.5.1.1.5.3.5 Exceptions/Errors

SCA443 The $sizeOf$ operation shall raise the CF::FileException when a file-related error occurs (e.g., file does not exist anymore).

3.1.3.5.1.1.5.4 close

3.1.3.5.1.1.5.4.1 Brief Rationale
The close operation is needed in order to release file resources once they are no longer needed.

3.1.3.5.1.1.5.4.2 Synopsis
void close() raises (FileException);

3.1.3.5.1.1.5.4.3 Behavior

SCA332 The close operation shall release any OE file resources associated with the component.

SCA333 The close operation shall make the file unavailable to the component.

3.1.3.5.1.1.5.4.4 Returns
This operation does not return any value.

3.1.3.5.1.1.5.4.5 Exceptions/Errors.

SCA334 The close operation shall raise the CF::FileException when it cannot successfully close the file.

3.1.3.5.1.1.5.5 setFilePointer

3.1.3.5.1.1.5.5.1 Brief Rationale
The setFilePointer operation positions the file pointer where the next read or write will occur.

3.1.3.5.1.1.5.5.2 Synopsis
void setFilePointer (in unsigned long filePointer) raises (InvalidFilePointer, FileException);

3.1.3.5.1.1.5.5.3 Behavior

SCA335 The setFilePointer operation shall set the filePointer attribute value to the input filePointer.

3.1.3.5.1.1.5.5.4 Returns
This operation does not return any value.

3.1.3.5.1.1.5.5.5 Exceptions/Errors

SCA336 The setFilePointer operation shall raise the CF::FileException when the file pointer for the referenced file cannot be set to the value of the input filePointer parameter.

SCA337 The setFilePointer operation shall raise the InvalidFilePointer exception when the value of the filePointer parameter exceeds the file size.

3.1.3.5.1.2 FileSystem

3.1.3.5.1.2.1 Description
The FileSystem interface defines operations that enable remote access to a physical file system (see Figure 3-49).
3.1.3.5.1.2.2 UML

```uml
+ SIZE: string = "SIZE" {readOnly}
+ AVAILABLE_SPACE: string = "AVAILABLE_SPACE" {readOnly}
+ CREATED_TIME_ID: string = "CREATED_TIME" {readOnly}
+ MODIFIED_TIME_ID: string = "MODIFIED_TIME" {readOnly}
+ LAST_ACCESS_TIME_ID: string = "LAST_ACCESS_TIME" {readOnly}
+ remove(string): void
+ copy(string, string): void
+ exists(string): boolean
+ list(string): FileInformationSequence
+ create(string): File
+ open(string, boolean): File
+ mkdir(string): void
+ rmdir(string): void
+ query(Properties*): void
```

**Figure 3-49: FileSystem Interface UML**

3.1.3.5.1.2.3 Types

3.1.3.5.1.2.3.1 UnknownFileSystemProperties.

The UnknownFileSystemProperties exception indicates a set of properties unknown by the component.

```c
exception UnknownFileSystemProperties { properties invalidProperties; }
```

3.1.3.5.1.2.3.2 fileSystemProperties Query Constants

Constants are defined to be used for the `query` operation (see section 3.1.3.5.1.2.5.9).

```c
const string SIZE = "SIZE";
const string AVAILABLE_SPACE = "AVAILABLE_SPACE";
```

3.1.3.5.1.2.3.3 FileInformationType

The FileInformationType indicates the information returned for a file. Not all the fields in the FileInformationType are applicable for all file systems. SCA338 At a minimum, the FileSystem interface implementation shall support name, kind, and size information for a file. Examples of other file properties that may be specified are created time, modified time, and last access time.

```c
struct FileInformationType
{
    string name;
    FileType kind;
    unsigned long long size;
    Properties fileProperties;
};
```
The name field of the FileInformationType struct indicates the simple name of the file. The kind field of the FileInformationType struct indicates the type of the file entry. The size field of the FileInformationType struct indicates the size in octets.

3.1.3.5.1.2.3.4 FileInformationSequence
The FileInformationSequence type defines an unbounded sequence of FileInformationTypes.

typedef sequence<FileInformationType> FileInformationSequence;

3.1.3.5.1.2.3.5 FileType
The FileType indicates the type of file entry. A file system may have PLAIN or DIRECTORY files and mounted file systems contained in a file system.

class FileType
{
    PLAIN,
    DIRECTORY,
    FILE_SYSTEM
};

3.1.3.5.1.2.3.6 CREATED_TIME_ID
The fileProperties field of the FileInformationType struct may be used to indicate the time a file was created. SCA445 For this property, the identifier is CREATED_TIME_ID and the value shall be an unsigned long long data type containing the number of seconds since 00:00:00 UTC, Jan 1, 1970.

const string CREATED_TIME_ID = "CREATED_TIME";

3.1.3.5.1.2.3.7 MODIFIED_TIME_ID
The fileProperties element of the FileInformationType struct may be used to indicate the time a file was last modified. SCA446 For this property, the identifier is MODIFIED_TIME_ID and the value shall be an unsigned long long data type containing the number of seconds since 00:00:00 UTC, Jan 1, 1970.

const string MODIFIED_TIME_ID="MODIFIED_TIME";

3.1.3.5.1.2.3.8 LAST_ACCESS_TIME_ID
The fileProperties element of the FileInformationType struct may be used to indicate the time a file was last accessed. SCA447 For this property, the identifier is LAST_ACCESS_TIME_ID and the value shall be an unsigned long long data type containing the number of seconds since 00:00:00 UTC, Jan 1, 1970.

const string LAST_ACCESS_TIME_ID="LAST_ACCESS_TIME";

3.1.3.5.1.2.4 Attributes
N/A.
3.1.3.5.1.2.5 Operations
3.1.3.5.1.2.5.1 remove
3.1.3.5.1.2.5.1.1 Brief Rationale
The remove operation provides the ability to remove a plain file from a file system.
3.1.3.5.1.2.5.1.2 Synopsis
void remove (in string fileName) raises (FileException, InvalidFileName);

3.1.3.5.1.2.5.1.3 Behavior
SCA339 The remove operation shall remove the plain file which corresponds to the input fileName parameter.

3.1.3.5.1.2.5.1.4 Returns
This operation does not return any value.

3.1.3.5.1.2.5.1.5 Exceptions/Errors
SCA340 The remove operation shall raise the CF::InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.
SCA341 The remove operation shall raise the CF::FileException when a file-related error occurs.

3.1.3.5.1.2.5.2 copy
3.1.3.5.1.2.5.2.1 Brief Rationale
The copy operation provides the ability to copy a plain file to another plain file.

3.1.3.5.1.2.5.2.2 Synopsis
void copy (in string sourceFileName, in string destinationFileName) raises (InvalidFileName, FileException);

3.1.3.5.1.2.5.2.3 Behavior
SCA342 The copy operation shall copy the source file identified by the input sourceFileName parameter to the destination file identified by the input destinationFileName parameter.
SCA343 The copy operation shall overwrite the destination file, when the destination file already exists and is not identical to the source file.

3.1.3.5.1.2.5.2.4 Returns
This operation does not return any value.

3.1.3.5.1.2.5.2.5 Exceptions/Errors
SCA344 The copy operation shall raise the CF::FileException exception when a file-related error occurs.
SCA345 The copy operation shall raise the CF::InvalidFileName exception when the destination pathname is identical to the source pathname.
SCA346 The copy operation shall raise the CF::InvalidFileName exception when the sourceFileName or destinationFileName input parameter is not a valid absolute pathname.

3.1.3.5.1.2.5.3 exists
3.1.3.5.1.2.5.3.1 Brief Rationale
The exists operation provides the ability to verify the existence of a file within a file system.

3.1.3.5.1.2.5.3.2 Synopsis
boolean exists (in string fileName) raises (InvalidFileName);
3.1.3.5.1.2.5.3.3 Behavior
SCA347 The exists operation shall check to see if a file exists based on the fileName parameter.

3.1.3.5.1.2.5.3.4 Returns
SCA348 The exists operation shall return TRUE if the file exists, or FALSE if it does not.

3.1.3.5.1.2.5.3.5 Exceptions/Errors
SCA349 The exists operation shall raise the CF::InvalidFileName exception when input fileName parameter is not a valid absolute pathname.

3.1.3.5.1.2.5.4 list
3.1.3.5.1.2.5.4.1 Brief Rationale
The list operation provides the ability to obtain a list of files along with their information in the file system according to a given search pattern. The list operation may be used to return information for one file or for a set of files.

3.1.3.5.1.2.5.4.2 Synopsis
FileInfoSequence list (in string pattern) raises (FileNotFoundException, InvalidFileName);

3.1.3.5.1.2.5.4.3 Behavior
SCA448 The list operation shall support the "*" and "?" wildcard characters (used to match any sequence of characters, including null, and any single character, respectively). SCA350 These wildcards shall only be applied following the right-most forward-slash character ("/") in the pathname contained in the input pattern parameter.

3.1.3.5.1.2.5.4.4 Returns
SCA351 The list operation shall return a FileInfoSequence for files that match the search pattern specified in the input pattern parameter. SCA352 The list operation shall return a zero length sequence when no file is found which matches the search pattern.

3.1.3.5.1.2.5.4.5 Exceptions/Errors
SCA353 The list operation shall raise the CF::InvalidFileName exception when the input pattern parameter is not an absolute pathname or cannot be interpreted due to unexpected characters.

SCA354 The list operation shall raise the CF::FileNotFoundException when a file-related error occurs.

3.1.3.5.1.2.5.5 create
3.1.3.5.1.2.5.5.1 Brief Rationale
The create operation provides the ability to create a new plain file on the file system.

3.1.3.5.1.2.5.5.2 Synopsis
File create (in string fileName) raises (InvalidFileName, FileNotFoundException);

3.1.3.5.1.2.5.5.3 Behavior
SCA355 The create operation shall create a new file based upon the input fileName parameter.

3.1.3.5.1.2.5.5.4 Returns
SCA356 The create operation shall return a file object reference to the created file.
3.1.3.5.1.2.5.5.5 Exceptions/Errors
SCA357 The create operation shall raise the CF::FileException if the file already exists or another file error occurred.
SCA358 The create operation shall raise the CF::InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.

3.1.3.5.1.2.5.6 open
3.1.3.5.1.2.5.6.1 Brief Rationale
The open operation provides the ability to open a plain file for read or write.

3.1.3.5.1.2.5.6.2 Synopsis
File open (in string fileName, in boolean read_Only) raises (InvalidFileName, FileException);

3.1.3.5.1.2.5.6.3 Behavior
SCA359 The open operation shall open the file referenced by the input fileName parameter.
SCA360 The open operation shall open the file with read-only access when the input read_Only parameter is TRUE. SCA361 The open operation shall open the file for write access when the input read_Only parameter is FALSE.

3.1.3.5.1.2.5.6.4 Returns
SCA362 The open operation shall return a FileComponent reference for the opened file.
SCA363 The open operation shall set the filePointer attribute of the returned file instance to the beginning of the file.

3.1.3.5.1.2.5.6.5 Exceptions/Errors
SCA364 The open operation shall raise the CF::FileException if the file does not exist or another file error occurred.
SCA365 The open operation shall raise the CF::InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.

3.1.3.5.1.2.5.7 mkdir
3.1.3.5.1.2.5.7.1 Brief Rationale
The mkdir operation provides the ability to create a directory on the file system.

3.1.3.5.1.2.5.7.2 Synopsis
void mkdir (in string directoryName) raises (InvalidFileName, FileException);

3.1.3.5.1.2.5.7.3 Behavior
SCA366 The mkdir operation shall create a file system directory based on the directoryName given. SCA367 The mkdir operation shall create all parent directories required to create the directoryName path given.

3.1.3.5.1.2.5.7.4 Returns.
This operation does not return any value.

3.1.3.5.1.2.5.7.5 Exceptions/Errors
SCA368 The `mkdir` operation shall raise the CF::FileException if the directory indicated by the input directoryName parameter already exists or if a file-related error occurred during the operation.

SCA369 The `mkdir` operation shall raise the CF::InvalidFileName exception when the directoryName is not a valid directory name.

3.1.3.5.1.2.5.8 `rmdir`

### Brief Rationale

The `rmdir` operation provides the ability to remove a directory from the file system.

### Synopsis

```c
void rmdir (in string directoryName) raises (InvalidFileName, FileException);
```

### Behavior

SCA370 The `rmdir` operation shall remove the directory identified by the input directoryName parameter.

SCA371 The `rmdir` operation shall not remove the directory identified by the input directoryName parameter when the directory contains files.

### Returns

This operation does not return any value.

### Exceptions/Errors

SCA372 The `rmdir` operation shall raise the CF::FileException when the directory identified by the input directoryName parameter does not exist, the directory contains files, or an error occurs which prohibits the directory from being deleted.

SCA373 The `rmdir` operation shall raise the CF::InvalidFileName exception when the input directoryName parameter is not a valid path prefix.

3.1.3.5.1.2.5.9 `query`

### Brief Rationale

The `query` operation provides the ability to retrieve information about a file system.

### Synopsis

```c
void query (inout Properties fileSystemProperties) raises (UnknownFileSystemProperties);
```

### Behavior

SCA374 The `query` operation shall return file system information to the calling client based upon the given fileSystemProperties' ID.

SCA440 The `FileSystem::query` operation shall recognize and provide the designated return values for the following fileSystemProperties (section 3.1.3.5.1.2.3.2):

1. `SIZE` - an ID value of "SIZE" causes the `query` operation to return an unsigned long long containing the file system size (in octets);
2. AVAILABLE_SPACE - an ID value of "AVAILABLE_SPACE" causes the query operation to return an unsigned long long containing the available space on the file system (in octets).

See section 3.1.3.5.1.2.3.2 for the constants for the FileSystemProperties.

3.1.3.5.1.2.5.9.4 Returns
This operation does not return any value.

3.1.3.5.1.2.5.9.5 Exceptions/Errors
SCA375 The query operation shall raise the UnknownFileSystemProperties exception when the given file system property is not recognized.

3.1.3.5.1.3 FileManager
3.1.3.5.1.3.1 Description
Multiple, distributed file systems may be accessed through a file manager. The FileManager interface, shown in Figure 3-50, appears to be a single file system although the actual file storage may span multiple physical file systems. This is called a federated file system. A federated file system is managed using the mount and unmount operations. Typically, the domain manager or system initialization software will invoke these operations.

The FileManager inherits the IDL interface of a FileSystem.
3.1.3.5.1.3.2 UML

![UML Diagram]

**Figure 3-50: FileManager Interface UML**

3.1.3.5.1.3.3 Types

3.1.3.5.1.3.3.1 MountType

The MountType structure identifies the file systems mounted within the file manager.

```cpp
struct MountType {
    string mountPoint;
    FileSystem fs;
};
```

3.1.3.5.1.3.3.2 MountSequence

The MountSequence is an unbounded sequence of MountTypes.

```cpp
typedef sequence <MountType> MountSequence;
```

3.1.3.5.1.3.3.3 NonExistentMount

The NonExistentMount exception indicates a mount point does not exist within the file manager.

```cpp
exception NonExistentMount{};
```
3.1.3.5.1.3.3.4 MountPointAlreadyExists
The MountPointAlreadyExists exception indicates the mount point is already in use in the file manager.

```c
exception MountPointAlreadyExists{};
```

3.1.3.5.1.3.3.5 InvalidFileSystem
The InvalidFileSystem exception indicates the FileSystem is a null (nil) object reference.

```c
exception InvalidFileSystem{};
```

### Attributes
N/A.

### Operations

#### 3.1.3.5.1.3.5.1 mount

**Brief Rationale**
The FileManager interface supports the notion of a federated file system. To create a federated file system, the mount operation associates a file system with a mount point (a directory name).

**Synopsis**
```c
void mount (in string mountPoint, in FileSystem file_System) raises (InvalidFileName, InvalidFileSystem, MountPointAlreadyExists);
```

**Behavior**

**SCA376** The mount operation shall associate the specified file system with the mount point referenced by the input mountPoint parameter.

- **SCA377** A mount point name shall begin with a "/" (forward slash character). The input mountPoint parameter is a logical directory name for a file system.

**SCA378** The mount operation shall raise the MountPointAlreadyExists exception when the mount point already exists in the file manager.

**SCA379** The mount operation shall raise the InvalidFileSystem exception when the input FileSystem is a null object reference.

**SCA461** The mount operation shall raise the CF::InvalidFileName exception when the input mount point does not conform to the file name syntax in section 3.1.3.5.2.2.3.

**SCA378** The mount operation shall raise the MountPointAlreadyExists exception when the mount point already exists in the file manager.

**SCA379** The mount operation shall raise the InvalidFileSystem exception when the input FileSystem is a null object reference.

**Returns.**
This operation does not return any value.

**Exceptions/Errors.**

**SCA461** The mount operation shall raise the CF::InvalidFileName exception when the input mount point does not conform to the file name syntax in section 3.1.3.5.2.2.3.

**Behavior**

#### 3.1.3.5.1.3.5.2 unmount

**Brief Rationale**
Mounted file systems may need to be removed from a file manager.

**Synopsis**
```c
void unmount (in string mountPoint) raises (NonExistentMount);
```

**Behavior**
SCA380 The *unmount* operation shall remove a mounted file system from the file manager whose mounted name matches the input mountPoint name.

3.1.3.5.1.3.5.2.4 Returns
This operation does not return any value.

3.1.3.5.1.3.5.2.5 Exceptions/Errors
SCA381 The *unmount* operation shall raise the NonExistentMount exception when the mount point does not exist.

3.1.3.5.1.3.5.3 getMounts

3.1.3.5.1.3.5.3.1 Brief Rationale
File management user interfaces may need to list a file manager's mounted file systems.

3.1.3.5.1.3.5.3.2 Synopsis
MountSequence getMounts();

3.1.3.5.1.3.5.3.3 Behavior
The *getMounts* operation returns a MountSequence that describes the mounted file systems.

3.1.3.5.1.3.5.3.4 Returns
SCA382 The *getMounts* operation shall return a MountSequence that contains the file systems mounted within the file manager.

3.1.3.5.1.3.5.3.5 Exceptions/Errors
This operation does not raise any exceptions.

3.1.3.5.1.3.5.4 query

3.1.3.5.1.3.5.4.1 Brief Rationale
The inherited *query* operation provides the ability to retrieve the same information for a set of file systems.

3.1.3.5.1.3.5.4.2 Synopsis
void query (inout Properties fileSystemProperties) raises (UnknownFileSystemProperties);

3.1.3.5.1.3.5.4.3 Behavior
SCA383 The *query* operation shall return the combined mounted file systems information to the calling client based upon the given input fileSystemProperties' ID elements. SCA441 As a minimum, the *query* operation shall support the following input fileSystemProperties ID elements:

1. SIZE - a property item ID value of "SIZE" causes the *query* operation to return the combined total size of all the mounted file system as an unsigned long long property value;

2. AVAILABLE_SPACE - a property item ID value of "AVAILABLE_SPACE" causes the *query* operation to return the combined total available space (in octets) of all the mounted file system as unsigned long long property value.

3.1.3.5.1.3.5.4.4 Returns
This operation does not return any value.

3.1.3.5.1.3.5.4.5 Exceptions/Errors

SCA384 The query operation shall raise the UnknownFileSystemProperties exception when the input FileSystemProperties parameter contains an invalid property ID element.

3.1.3.5.2 Components

Framework Services Components provide general software capabilities (not directly associated with logical devices) that will be utilized by platform developers.

The File Services (FileComponent, FileSystemComponent and FileManagerComponent) consist of interfaces and components that are used to manage and access a potentially distributed file system. The File Services are used for installation and removal of application and artifact files within the system, and for loading and unloading those files on the various processors that they execute upon.

3.1.3.5.2.1 FileComponent

3.1.3.5.2.1.1 Description

The FileComponent provides the ability to read and write files residing within a file system.

![Figure 3-51: FileComponent UML](image)

3.1.3.5.2.1.2 Associations

- physicalFile: A FileComponent is the logical proxy for a physical file that resides on the actual file system.

3.1.3.5.2.1.3 Semantics

SCA397 A FileComponent's filePointer attribute shall be set to the beginning of the file when a FileComponent is opened for read only or created for the first time. SCA398 A FileComponent's filePointer attribute shall be set at the end of the file when a FileComponent already exists and is opened for write.
3.1.3.5.2.1.4 Constraints
SCA399 A FileComponent shall realize the File interface.

3.1.3.5.2.2 FileSystemComponent

3.1.3.5.2.2.1 Description
A FileSystemComponent realizes the FileSystem interface, may be associated with a FileManagerComponent and consists of many FileComponents. The FileSystemComponent provides a container for managing the lifespan and organization of FileComponents.

```
+ remove(string): void
+ copy(string, string): void
+ exists(string): boolean
+ list(string): FileInformationSequence
+ create(string): File
+ open(string, boolean): File
+ mkdir(string): void
+ rmdir(string): void
+ query(Properties*): void
```

```
FileComponent «manages» +fileComponent 0..*
```

**Figure 3-52: FileSystemComponent UML**

3.1.3.5.2.2.2 Associations
- fileComponent: A FileSystemComponent manages the creation, deletion and manipulation of FileComponents within a file system.

3.1.3.5.2.2.3 Semantics
The files stored on a file system may be plain files or directories. SCA400 Valid characters for a FileSystemComponent file name and file absolute pathname shall adhere to POSIX compliant file naming conventions. Valid characters for a filename or directory name are the 62 alphanumeric characters (Upper, and lowercase letters and the numbers 0 to 9) in addition to the "." (period), "_" (underscore) and "-" (hyphen) characters. The filenames "." ("dot") and "." ("dot-dot") are invalid in the context of a file system. Valid pathnames include the "/" (forward slash) character in addition to the valid filename characters. A valid pathname may consist of a single filename.

3.1.3.5.2.2.4 Constraints
SCA401 A FileSystemComponent shall realize the FileSystem interface. SCA402 Valid individual filenames and directory names for a FileSystemComponent shall be 40 characters or less. SCA403 A valid pathname for a FileSystemComponent shall not exceed 1024 characters.
3.1.3.5.2.3 FileManagerComponent

3.1.3.5.2.3.1 Description

The FileManagerComponent extends a FileSystemComponent by adding the capability to allow multiple, distributed FileSystemComponents to be accessed through a FileManagerComponent. The FileManagerComponent appears as a single file system although the actual file storage may span multiple physical file systems. A FileManagerComponent implements the inherited FileSystem operations defined in section 3.1.3.5.1.2 for each mounted FileSystemComponent.

3.1.3.5.2.3.2 Associations

- filesystem: A FileManagerComponent manages the mounting and un-mounting of its contained FileSystemComponent(s).

3.1.3.5.2.3.3 Semantics

Based upon the pathname of a directory or file and the set of mounted file systems, the FileManagerComponent delegates the FileSystemComponent operations to the appropriate file system. For example, if a file system is mounted at "/ppc2", an open operation for a file called "/ppc2/profile.xml" would be delegated to the mounted file system. The mounted file system will be given the filename relative to it. In this example the FileSystemComponent's open operation would receive "/profile.xml" as the fileName argument.

Another example of this concept is shown using the copy operation. When a client invokes the copy operation, the FileManagerComponent delegates the operation to the appropriate FileSystemComponents (based upon supplied pathnames) thereby allowing copy of files between FileSystemComponents.

If a client does not need to mount and unmount FileSystemComponents, it may treat the FileManagerComponent as a FileSystemComponent by widening a FileManager interface reference to a FileSystem interface reference (because the FileManager interface is derived from a FileSystem interface).

The FileSystem operations ensure the filename/directory arguments given are absolute pathnames relative to a mounted file system. SCA404 The FileSystem operations realized by a
FileManagerComponent shall remove the name of the mounted file system from input pathnames before passing the pathnames to any operation on a mounted file system. SCA405 A FileManagerComponent shall propagate exceptions raised by a mounted file system. SCA406 A FileManagerComponent shall use the FileSystem operations of the FileSystemComponent whose associated mount point exactly matches the input fileName parameter to the lowest matching subdirectory.

The system may support multiple FileSystemComponents. Some file systems correspond directly to a physical file system within the system. A FileManagerComponent supports a federated, or distributed, file system that may span multiple FileSystemComponents. From the client perspective, the FileManagerComponent may be used just like any other FileSystemComponent since the FileManagerComponent inherits all the FileSystem operations.

3.1.3.5.2.3.4 Constraints
SCA408 A FileManagerComponent shall realize the FileManager interface.
SCA409 A FileManagerComponent instantiation shall fulfill the FileSystemComponent component requirements.

3.1.3.5.2.4 BasePlatformComponent
3.1.3.5.2.4.1 Description
A BasePlatformComponent is an abstract component utilized by the SCA Base Device Components and Framework Services Components.

![Figure 3-54: BasePlatformComponent UML](image)

3.1.3.5.2.4.2 Associations
- fullRegistrar: A BasePlatformComponent may be associated with a full registrar for unregistering itself.

3.1.3.5.2.4.3 Semantics
A BasePlatformComponent is not limited to using the services designated as mandatory by Appendix B and thus may use any service provided by the OE. SCA298 A BasePlatformComponent shall register with its DeviceManagerComponent via the ComponentRegistry::registerComponent operation. A BasePlatformComponent determines its
DeviceManagerComponent using its input execute parameters or by other techniques if it was not deployed by a DeviceManagerComponent. The CF::ComponentType's type field is a BasePlatformComponent type (i.e., DEVICE_COMPONENT, LOADABLEDEVICE_COMPONENT, EXECUTABLEDEVICE_COMPONENT, SERVICE_COMPONENT, MANAGEABLESERVICE_COMPONENT and PLATFORM_COMPONENT_FACTORY_COMPONENT). A BasePlatformComponent may add its allocation properties to the CF::ComponentType's specializedInfo field. If the BasePlatformComponent does not add its allocation properties, then the DeviceManagerComponent is responsible for adding them.

3.1.3.5.2.4.4 Constraints
SCA565 A BasePlatformComponent shall fulfill the BaseComponent requirements.
Each BasePlatformComponent is accompanied by an SPD file per section 3.1.3.6.1.
A BasePlatformComponent may unregister itself via the FullComponentRegistry interface using the Component Registry execute parameter.

3.1.3.5.2.5 PlatformComponentFactoryComponent
3.1.3.5.2.5.1 Description
A platform component factory is an optional mechanism that may be used to create device or service components.

<table>
<thead>
<tr>
<th>BasePlatformComponent</th>
<th>BaseFactoryComponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3-55: PlatformComponentFactoryComponent UML**

3.1.3.5.2.5.2 Associations
N/A.
3.1.3.5.2.5.3 Semantics
A PlatformComponentFactoryComponent is used to create a service or device component. A DeviceManagerComponent is not required to use platform component factories to create devices and services. A DCD specifies which PlatformComponentFactoryComponents are to be used by the DeviceManagerComponent.
3.1.3.5.2.5.4 Constraints

SCA527 A PlatformComponentFactoryComponent instantiation shall fulfill the BaseFactoryComponent requirements.

SCA567 A PlatformComponentFactoryComponent instantiation shall fulfill the BasePlatformComponent requirements. SCA416 The PlatformComponentFactoryComponent shall only deploy DeviceComponents or ServiceComponents.

3.1.3.5.2.6 ServiceComponent

3.1.3.5.2.6.1 Description

A ServiceComponent is a platform software component that can implement any interface(s) and does not manage hardware. A ServiceComponent usually comes into existence at platform startup. The SCA identifies its services in section 3.1.2, defining some such as event and lightweight log, which are known as the SCA Services. ServiceComponents do not implement the SCA Base Application interfaces and may use any operating system APIs provided by the OE (i.e. they are not restricted to using only the APIs as specified in Appendix B).

![ServiceComponent UML](image)

**Figure 3-56: ServiceComponent UML**

3.1.3.5.2.6.2 Associations

N/A.

3.1.3.5.2.6.3 Semantics

ServiceComponents are typically used by AssemblyComponents but there is nothing restricting platform services being utilized by any type of BasePlatformComponent (e.g. DeviceComponents).

SCA314 All ServiceComponents started up by a DeviceManagerComponent shall have a handler registered for the POSIX SIGQUIT signal.

A ServiceComponent may realize an interface directly without ports, similar to an SCA service, or have ports. SCA317 The values associated with the parameters (SERVICE_NAME) as described in 3.1.3.3.2.4.3 shall be used to set the platform service's ComponentIdentifier interface identifier attribute. The CF::ComponentType's type field of the registering component is assigned as a SERVICE_COMPONENT.

3.1.3.5.2.6.4 Constraints

A ServiceComponent instantiation is a BasePlatformComponent. SCA460 Each ServiceComponent shall have an SPD as described in section 3.1.3.6.4. The ServiceComponent
abstraction can represent Service_Components implemented by a third party (e.g. commercial) provider. In those cases the implementation oriented ServiceComponent requirements are not applicable. SCA568 A ServiceComponent shall fulfill the BasePlatformComponent requirements.

3.1.3.5.2.7 ManageableServiceComponent
3.1.3.5.2.7.1 Description
A ManageableServiceComponent extends the ServiceComponent by adding support for the SCA Base Application interfaces.

![ManageableServiceComponent UML](image)

**Figure 3-57: ManageableServiceComponent UML**

3.1.3.5.2.7.2 Associations
N/A.
3.1.3.5.2.7.3 Semantics
The CF::ComponentType's type field of the registering component is a MANAGEABLE_SERVICE_COMPONENT.
3.1.3.5.2.7.4 Constraints
SCA530 A ManageableServiceComponent shall fulfill the ServiceComponent requirements. SCA569 A ManageableServiceComponent shall realize the LifeCycle interface.

3.1.3.6 Domain Profile

The hardware devices and software components that make up an SCA system domain are described by a set of files that are collectively referred to as a Domain Profile. These files describe the identity, capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up the system. All of the descriptive data about a system is expressed in the descriptor vocabulary.

The types of descriptor files that are used to describe a system's hardware and software assets are depicted in Figure 2-7. The vocabulary within each of these files describes a distinct aspect of the hardware and software assets. The collection of descriptor files which are associated with a particular software component is referred to as that component's software profile. The contents of a profile depends on the component being described, although every profile contains a SPD - all profiles for components contain a SCD. A software profile for an application contains a
SAD, the device manager profile contains a Device Configuration Descriptor (DCD), and the domain manager software profile contains a DMD.

SCA463 Domain Profile files shall be compliant to the descriptor files provided in Appendix D.

3.1.3.6.1 **Software Package Descriptor (SPD)**
An SPD identifies a software component implementation(s). General information about a software package, such as the name, author, property file, and implementation code information and hardware and/or software dependencies are contained in an SPD file.

3.1.3.6.2 **Software Component Descriptor (SCD)**
An SCD contains information about a specific SCA software component. An SCD file contains information about the interfaces that a component provides and/or uses.

3.1.3.6.3 **Software Assembly Descriptor (SAD)**
An SAD contains information about the components that make up an application. The application factory uses this information when creating an application.

3.1.3.6.4 **Properties Descriptor (PRF)**
A PRF contains information about the properties applicable to a software package or a device package. A PRF contains information about the properties of a component such as configuration, test, execute, and allocation types.

3.1.3.6.5 **Device Package Descriptor (DPD)**
A DPD identifies a class of a device. A DPD also has properties that define specific properties (capacity, serial number, etc.) for this class of device. The use of the DPD is optional within a system, however if it is used the reference to this file will be made from the DCD file.

3.1.3.6.6 **Device Configuration Descriptor (DCD)**
A DCD contains information about the devices associated with a device manager, how to find the domain manager, and the configuration information for the components that it deploys.

3.1.3.6.7 **Domain Manager Configuration Descriptor (DMD)**
A DMD contains configuration information for the domain manager.

3.1.3.6.8 **Platform Deployment Descriptor (PDD)**
A PDD identifies the logical relationships between platform resources within the OE's registered services and devices. The use of the PDD is optional within a system, however if it is used the reference to this file will be made from the DMD file. A PDD file may be used to exert a greater degree of control over the application deployment process. The file contains information that describes the composition (i.e. included services and devices) of virtual channels within a platform domain.

3.1.3.6.9 **Application Deployment Descriptor (ADD)**
An ADD contains precedence lists that are used for deploying application instances within a platform domain. The use of the ADD is optional within a system, however if it is used the reference to this file will be made from a SAD file. An ADD file contains application names and references the virtual channels defined in the PDD file.
4 CONFORMANCE

SCA conformance is achieved when a product successfully implements all applicable requirements identified within the scope of its declared conformance statement. Language used to identify requirements within this specification is defined in section 1.2.2. Requirements stated in this specification take precedence when they are in conflict with other existing standards/specifications, cited or not cited.

The JTNC is the Specification Authority (SA) responsible for developing, maintaining, evolving and interpreting this standard.

4.1 CONFORMANCE CRITERIA

SCA conformance language is referenced in several parts of the specification:

- The SCA technology independent model representation (i.e. Platform Independent Model) is summarized in this specification.
  - The SCA technology independent model comprises a set of interfaces and component definitions that are appropriate for building SCA products (e.g. applications, devices and services). SCA products may be realized using a variety of technologies (e.g., CORBA, JAVA, MHAL Communications Service, etc.).
- The SCA technology specific model representations (i.e. Platform Specific Models) are defined in the corresponding SCA Appendix.
  - The SCA technology specific models comprise of technology specific mappings, transformations, and model representations used in the realization of the technology on a specific platform.

SCA Conformance can be achieved using multiple methods. Therefore, several separate conformance points are defined below.

4.1.1 Conformance on the Part of an SCA Product

The interfaces and components of this specification are not required to be used solely for a particular platform or application. An SCA product uses the interfaces and component definitions that meet their needs.

Conformance for an SCA Product is at the level of usage as follows:

- A technology independent representation of an interface defined in this specification needs to be conformant with an identified Profile or collection of UOFs as described in Appendix F;
- A technology specific representation (no matter what language) of an interface defined in this specification needs to be conformant (signature equivalent) to the technology independent interface definition as described in this specification;
- A technology specific implementation (no matter what language) of a component defined in this specification needs to be conformant to both the component technology independent representation (e.g. semantics, ports, interfaces, properties) and any associated technology specific semantics and interface definitions as described in this specification.
For example, a component is considered to be conformant to the Component Framework CORBA/XML platform if it does all of the following:

- Realizes the OMG IDL defined interfaces referred to by the SCA CORBA component representation;
- Implements XML Domain Profile file serialization in accordance with the format defined in Appendix D;
- Implements the behavioral requirements identified by the component stereotype's technology independent representation.

Note that the semantics for an interface identified in the component technology independent representation are defined by the interface signature, associations and semantics of the corresponding interface in the SCA technology independent representation.

4.1.2 Conformance on the Part of an SCA OE component

The SCA OE contains the requirements of the operating system, transfer mechanism, and the CF interfaces and operations. Conformance of the CF elements is governed by the SCA Product rules defined in Section 2.2; however the other OE elements (i.e. those without components or interfaces defined within the main body of the specification) are subject to a unique conformance rule.

Conformance for an OE element (without a corresponding component or interface definition) is at the level of usage as follows:

- Realizes the applicable interfaces associated with its OE capability, or the interfaces associated with a documented profile of that capability;
- Implements the applicable behavioral requirements defined within its capability description (i.e. the corresponding SCA Appendix).

Thus, an OE implementation as defined in this specification could provide support for an AEP POSIX layer per Appendix B; and provide distributed communications in accordance with the CORBA (full profile) per Appendix E.

4.2 SAMPLE CONFORMANCE STATEMENTS

An SCA product can be identified as being conformant to a specific version of the SCA and the specific technology that the product realizes.

- "Product A is an SCA conformant waveform for the CORBA/XML platform."
- "Product B is an SCA conformant Audio Device for the J2EE/XML platform."
- "Product C is an SCA conformant Core Framework for the CORBA/XML platform."
- "Product D is an SCA conformant Operating Environment containing a lightweight AEP conforming POSIX layer and a CORBA (full profile) transfer mechanism."