SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION

FINAL / 15 May 2006

Version 2.2.2

Prepared by:

JTRS Standards
Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS)
Space and Naval Warfare Systems Center San Diego
53560 Hull Street, San Diego CA 92152-5001

REVISION SUMMARY

Version	Revisions
1.0	Formal release for initial validation.
1.1	Incorporate approved Change Proposals, numbers 97, 99, 110, 160, 161, 162, 164, 171, 177, 178, 179, 180, 193, 195, 201, 204, 205, 208, 209, 211, 216.
2.0	Incorporate approved Change Proposals, numbers 39, 105, 119, 147, 175, 186, 191, 192, 210, 217, 218, 219, 220, 222, 223, 225, 226, 227, 229, 231, 232, 235, 237, 240, 243, 249, 255, 258, 266, 270, 275, 276, 277, 278, 282, 283, 285, 299, 307, 308, 310, 311, 332, 335, 336, 337, 341, 342, 343, 344, 345.
2.1	Incorporate approved Change Proposals, numbers 88, 102, 142, 306, 316, 353, 357, 358, 359, 360, 365, 366, 367, 369, 370, 371, 372, 373, 419, 468, 471, 472, 473, 475, 476, 477
2.2	Incorporate approved Change Proposals, numbers 138, 250, 279, 338, 388, 466, 486, 487, 488, 495, 497, 504, 508, 509, 513, 514, 515, 517
2.2.1	Incorporate approved Change Proposals, SCA-CCM* numbers 1, 4, 5, 13, 15, 20, 23, 26, 28, 29, 30, 44, 70, 74, 78, 77, 100, 102, 107,
2.2.2	Incorporate approved Change Proposals: SCA-CCM 001, 004, 005, 013, 015, 020, 022, 023, 024, 025, 026, 027, 028, 029, 030, 044, 070, 074, 077, 078, 083, 084, 087, 091, 095, 097, 100, 102, 104, 107, 108, 118, 120, 121, 122, 123, 124, 125, 134, 137, 140, 141, 142, 145, 149, 152, 153, 178, 182, 188, 189, 190, 194, 202, 234, 264, 273, 276, 283, 284, 299, 303, 307, 308, 309, 313, 314, 316, 318, 319, 320, 321, 323, 324, 325

* The numbering scheme for change proposals was changed for version 2.2.1.

TABLE OF CONTENTS

LIST OF FIGURES

FOREWORD

Introduction. The Software Communication Architecture (SCA) is published by the Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS). This architecture was developed to assist in the development of software defined radio 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 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,
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.

Support and Rationale Document (SRD). The Support and Rationale document (SRD) provides the rationale used to determine the requirements contained in this document. The SRD also contains further supporting material including historical references, examples, and implementation considerations and should be consulted when attempting to develop a product compliant with this specification.

Feedback. An open architecture framework is greatly improved through active feedback and recommended changes from a wide audience of potential users. The JPEO JTRS solicits and encourages feedback to this document and provides a website for submitting feedback and change proposals. The website can be found at https://jtrs.spawar.navy.mil/sca . Change proposals to the SCA shall be unencumbered by copyrights, export restrictions, or intellectual property rights.

1 INTRODUCTION

The Software Communications Architecture (SCA) establishes an implementation-independent framework with baseline requirements for the development of software for software defined radios. The SCA is an architectural framework that was created to maximize portability, interoperability, and configurability of the software 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. The SCA addresses a portion of software portability and interoperability concerns – other aspects of these properties are addressed by different means as indicated in Table 1.

Table 1: Portability and Interoperability and the SCA

Goals	Responsibility
Software (operating on the host environment) meets all original performance specifications; interoperate over the air (OTA) with other communication systems, and not conflict with the correct operation of other software when deployed on a SCA compliant system.	System Engineering and Testing
Software (compiled for a target host environment) may be installed, configured and operated on different SCA compliant operating environments with a minimal amount of changes to the original code.	Software Communications Architecture
Software implementations may be moved from one specific host or development environment (i.e. a specific set of compilers, linkers, libraries, OS, chipsets, etc.) to another with a minimal amount of changes to the original code.	Coding Standards, Software Architecture and Design

The main body of the SCA addresses the goals identified for it in Table 1, while appendices to this specification are used to extend the scope of the SCA in order to address some of the desired portability and interoperability characteristics identified elsewhere in the table.

1.2 COMPLIANCE

As the Certification Authority, the JPEO JTRS is the sole entity that may authorize the use of any trademarks, certification markings, as well as verbal or written claims with respect to a product's compliance to this specification. Specific authorities and certification requirements are found in section 4.

Compliance to this specification requires a product to meet all applicable requirements identified within the scope of the specification. Applicability of requirements to specific products is determined by the Certification Authority. Language used to identify requirements within this specification is defined in section 1.3.1.5. Requirements stated in this specification take precedence when they are in conflict with other existing standards/specifications, cited or not cited.

1.3 DOCUMENT CONVENTIONS, TERMINOLOGY, AND DEFINITIONS

1.3.1 Conventions and Terminology

1.3.1.1 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.

1.3.1.2 Unified Modeling Language

The Unified Modeling Language (UML) [2], defined by the Object Management Group (OMG), is used to graphically represent SCA interfaces, operational scenarios, use cases, and collaboration diagrams. Where feasible, the UML used in this specification conforms to the syntax recommended by the OMG for Common Object Request Broker Architecture (CORBA) usage [A].

1.3.1.3 Interface Definition Language

The OMG defined Interface Definition Language (IDL), [E] is used to define the SCA interfaces within this specification.

1.3.1.4 eXtensible Markup Language

eXtensible Markup Language (XML) [3] 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. The term "profile" is

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 file (e.g. SAD, DCD) refer to the raw XML format per the definitions in section 3.1.3.5.

1.3.1.5 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 a truly 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.

1.3.1.6 Core Framework Interface 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 realized interfaces are presented in plain text. "CF" precedes references to Core Framework Base Types (3.1.3.6)

1.3.1.7 Figures

The figures contained in this document use coloration to identify elements of the SCA or how an object in a figure relates to those elements. Brown is used to indicate elements of the OS, orange for the Framework Control, Framework Service, and Device Interfaces and yellow for the Base Application Interfaces. Figure objects containing more than one of these colors indicate that the object relates to more than one SCA element – usually depending on context.

1.3.2 Definitions

A list of acronyms and definitions used in this specification are provided in Appendix A.

1.4 DOCUMENT CONTENT

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

Section 2 provides an overview of the Software Communications Architecture as well as a description of the interfaces and behaviors prescribed by the specification.

Section 3 provides the detailed description of the architecture framework and the specification requirements.

Section 4 defines the appropriate authorities for incorporating changes, recommendations, additions, or retractions into this specification, for validating compliance, and for granting certification.

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

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

Appendix C contains the Interface Definition Language (IDL) code used to define the interfaces required by this specification.

Appendix D contains the definitions and requirements for creating the SCA Domain Profile.

1.5 NORMATIVE REFERENCES

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

[1] Information technology - Portable Operating System Interface (POSIX®), ISO/IEC 9945:2003

[2] UML: OMG (Object Management Group) Unified Modeling Language Specification, Version 1.4.2, formal/05-04-01 (also available as ISO/IEC 19501:2005(E)

[3] XML: Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 04 February 2004F

[4] IEEE Standard for Information Technology – Standardized Application Environment Profile (AEP) – POSIX® Realtime and Embedded Application Support, IEEE Std 1003.13-2003.

[5] Minimum CORBA Specification version 1.0: OMG Document formal/02-08-01, August 2001.

[6] OMG Document formal/00-11-01: Interoperable Naming Service Specification.

[7] OMG Lightweight Log Service Specification: OMG Document formal/05-02-02: v1.1

[8] OMG Event Service Specification: OMG Document formal/01-03-01 and Event Service IDL, v1.1.

[9] DCE UUID standard (OSF Distributed Computing Environment, DCE 1.1 Remote Procedure Call).

1.6 INFORMATIVE REFERENCES

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

[A] OMG Document formal/02-04-01; UML Profile for CORBA, version 1.0.

® POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

[B] "Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides.

[C] Joint Program Executive Office for the Joint Tactical Radio System (JPEO JTRS), JTRS Charter 13 October 2005.

[D] JTRS Standards Waiver Process, JPEO JTRS, 2 December 2005, version 1.0.

[E] The Common Object Request Broker: Architecture and Specification, version 3.0.3 formal/04-03-12, Object Management Group, Inc. (OMG)

[F] Joint Technical Architecture, Version 2.0, 26 May 1998.

[G] C Standard: Programming languages – C, ISO/IEC 9899:1999.

[H] ISO/IEC 10731 Conventions for the Definition of OSI Services, Annex D Alternative and Additional Time Sequence Diagrams for Two-party Communications.

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 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.

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 components implementing that interface. Within this specification, the Unified Modeling Language (UML) [2] is used to graphically represent interfaces and the Interface Definition Language (IDL) provided in Appendix C contains the textual representation of the interfaces.

2.2 ARCHITECTURE OVERVIEW

2.2.1 Goals and Context

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 software defined radios (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 redeployable 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.

Since the functionality of software itself is virtually limitless, there is a large degree of 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 also 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.

2.2.2 Core Framework

The Core Framework is the essential set of open application-layer CORBA interfaces and services which provide an abstraction of the underlying system software and hardware. The Core Framework consists of:

Base Application Interfaces: *Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory,* and *Resource*), which provide the management and control interfaces for all system software components.

Base Device Interfaces: *Device, LoadableDevice, ExecutableDevice,* and *AggregateDevice,* which allows the management and control of hardware devices within the system through their software interface,

Framework Control Interfaces: *Application, ApplicationFactory, DomainManager,* and *DeviceManager,* which control the instantiation, management, and destruction/removal of software from the system,

Framework Services Interfaces: *File, FileSystem, and FileManager,* that provide additional support functions and services.

2.2.3 Definitions

The SCA differentiates between waveform "application" software – software that manipulates input data and determines the output of the system – from the software that provides the capabilities for waveforms to execute and access to the systems hardware resources. The "application" software implements the Base Application Interfaces identified in section 2.2.2. The software components that provide access to the system hardware resources are referred to as SCA "devices" that implement the Base Device Interfaces. Non-hardware (software-only) resources provided by the system for use by applications are generically referred to as "services", however the SCA does not specify an interface that must be realized by these components. The SCA standardizes the component interfaces but does not place implementation requirements (e.g. transport mechanisms) on the software.

The software components which provide for the management and execution of the SCA applications and devices comprise the SCA-defined operating environment (OE). The OE consists of an operating system (OS), CORBA middleware (including the OMG-defined Event

and Naming Services), and the elements defined by the Framework Control and Service Interfaces.

2.2.4 Structure

The architectural structure of the SCA is presented in Figure 2-1. In the SCA, an application consists of multiple software components that are loaded onto a distributed-processing system. These components are managed by an implementation of the Framework Control Interfaces. The application components communicate either with each other or with the services and devices provided by the system through extensions of the SCA-defined *Port* interface. Similarly, communications between the application and the Framework Services Interfaces are accomplished through the CORBA middleware. It is intended that the APIs to the services and devices ("System Components" in Figure 2-1) be standardized for a given system or domain so that in conjunction with the Framework Interfaces, all communications between the application and the system are uniform across multiple systems. However, being system and domain specific, the standardization of these interfaces is outside the scope of this specification.

An application may access OS functionality but is restricted to the operations enumerated in the SCA Application Environment Profile (Appendix B) which is a subset of the Portable Operating System Interface (POSIX) specification [4]. POSIX is an accepted industry standard and its real-time extensions are compatible with the requirements to support the OMG CORBA specification. Since defined POSIX profiles can encompass more features than are necessary to control a typical implementation, this specification defines a minimal POSIX profile to facilitate attainment of the SCA objectives.

Similar to the application components, system components are managed by the Framework Control Interfaces through the Base Device Interfaces and are limited . However, unlike application components, system components are not restricted in their use of functionality provided by the OS since these components are in general, system specific.

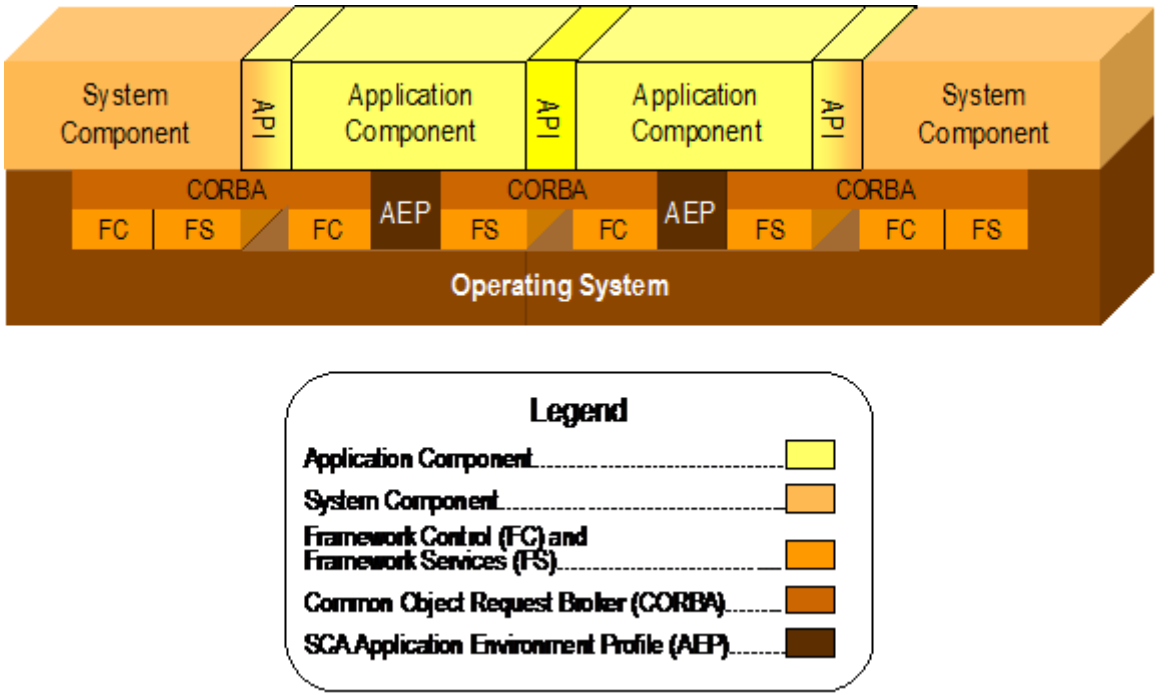

Figure 2-1: SCA Architecture Layer Diagram

All SCA compliant systems require certain software components to be present in order to provide for component deployment, management, and interconnection. These components include the *DomainManager* (including support for the *ApplicationFactory* and *Application* interfaces), *DeviceManager*, *FileManager*, and *FileSystem* interfaces and their required behaviors. The management hierarchy of these entities is depicted in Figure 2-2.

An SCA compliant system includes a domain manager 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, and all applications (and their resources).

Each device manager, in turn, contains complete knowledge of a set of devices and/or services. A system may have multiple device managers but each device manager registers with the domain manager to ensure that the domain manager has complete cognizance of the system. A device manager may have an associated file system (or file manager to support multiple file systems) as indicated in the Figure 2-2.

The implementation of the *Application* interface (created by the *ApplicationFactory*) OE provided proxy for an application contains all the information regarding a specific application that is instantiated on the system.

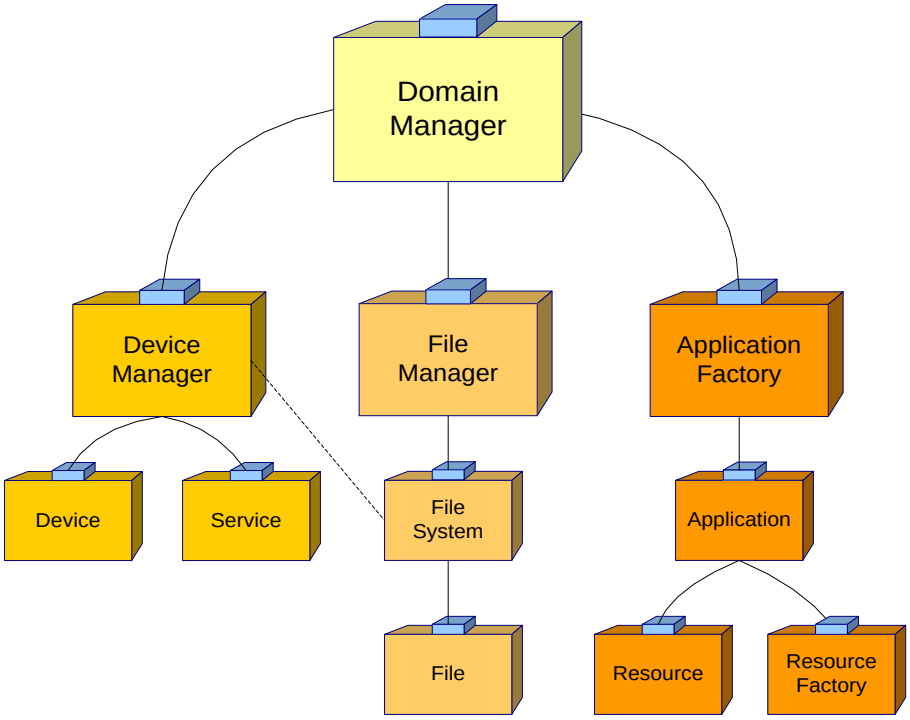

Figure 2-2: SCA Management Hierarchy at Instantiation

In order to describe the characteristics and attributes of the services, devices, and applications installed on the system, the SCA defines a set of files referred to as the Domain Profile. The Domain Profile is a hierarchical collection of eXtensible Markup Language (XML) files that define the properties of all software components in the system. All CORBA software elements of the system are described by a Software Package Descriptor (SPD) and a Software Component Descriptor (SCD) file.

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.), configuration and initialization properties (contained in a Properties File), dependencies to other SPDs and devices, and a reference to a Software Component Descriptor.

The Software Component Descriptor (SCD) defines CORBA interfaces supported and used by a specific component.

Since applications are composed of multiple SW components a Software Assembly Descriptor (SAD) file is defined to determine the composition and configuration of the application. The SAD references all SPDs 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 a single component within the application as the assembly controller.

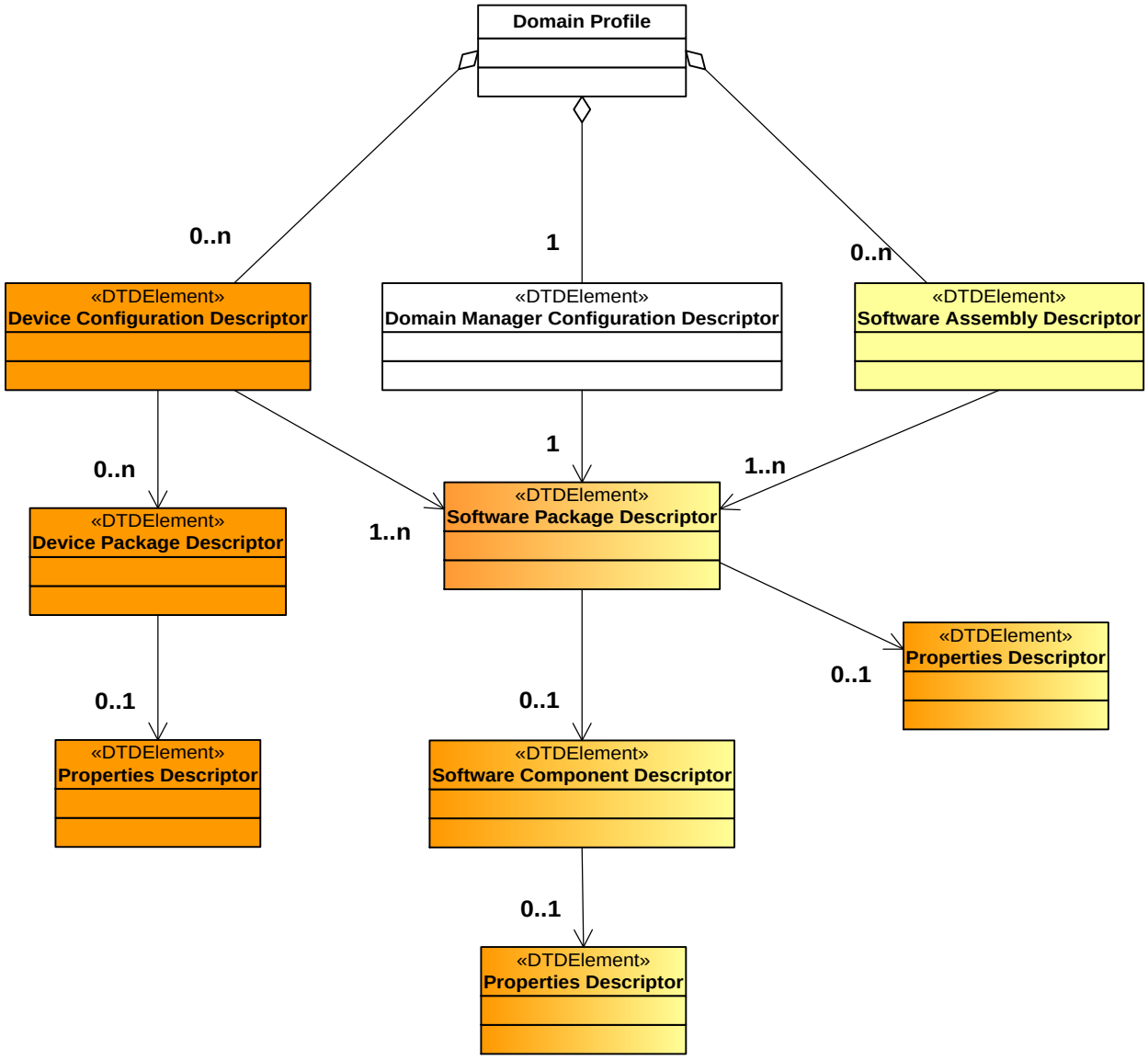

Figure 2-3: Relationship of Domain Profile XML File Types

An application consists of one or more software modules that, when loaded and executed, create one or more components (e.g. *Resources* or *ResourceFactories*), which comprise the application. These components use the facilities of the platform devices and services. The software profile for an application consists of one SAD file that references (directly or indirectly) one or more SPD, SCD, and properties (PRF) files. An SPD file contains the details of an application's software module that is to be loaded and executed. The SPD specifies the *Device* implementation requirements for loading dependencies (processor kind, etc.) and processing capacities (e.g., memory, process) for an application software module.

Similar to the application SAD, a device manager has an associated Device Configuration Descriptor (DCD) file. The DCD identifies all devices and services associated with this device manager, by referencing the 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. In addition to an SPD, a device may have a Device Package Descriptor (DPD) file which provides a description of the hardware device associated with this (logical) device including description, model, manufacturer, etc.

The implementation of the Domain Manager is itself described by the DomainManager Configuration Descriptor (DMD) which provides the location of the (SPD) file for the specific *DomainManager* implementation to be loaded. It also specifies the connections to other software components (services and devices) which are required by the domain manager.

2.2.4.1 Bus Layer (Board Support Package)

The SCA is capable of operating on commercial bus architectures. The OE supports reliable transport mechanisms, which may include error checking and correction at the bus support level.

2.2.4.2 Network & Serial Interface Services

The SCA relies on commercial components to support multiple unique serial and network interfaces. To support these interfaces, various low-level network protocols may be used. Elements of waveform networking functionality may also exist at the Operating System layer.

2.2.4.3 Operating System

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

2.2.4.4 CORBA Middleware

CORBA is used as the message passing technique for the distributed processing environment. CORBA is a cross-platform framework that is used to standardize client/server operations when using distributed processing. Distributed processing is a fundamental aspect of the system architecture and CORBA is a widely used “Middleware” service for providing distributed processing.

2.2.4.5 Applications

Applications consist of one or more resources. The *Resource* interface provides a common SCA API for the control and configuration of software components. Application developers may extend these capabilities by creating specialized *Resource* interfaces for the application. At a minimum, the extension inherits the *Resource* interface. The design of a resource’s internal functionality is not dictated by the Software Communications Architecture. This is left to the application developer.

2.2.4.5.1 Adapters

Adapters are resources or devices used to support the use of non-CORBA capable elements within the domain. Adapters are used in an implementation to provide the translation between non-CORBA-capable components or devices and CORBA-capable *Resources*. The Adapter

concept is based on the industry-accepted Adapter design pattern [B]. Since an Adapter implements the CF CORBA interfaces known to other CORBA-capable *Resources*, the translation service is transparent to the CORBA-capable *Resources*. Adapters become particularly useful to support non-CORBA-capable processing elements.

2.2.4.6 Reference Model

The SCA realizes the reference model by defining a standard unit of functionality called a *Resource*. All applications are comprised of resources and using devices. Specific resources and devices can be identified corresponding to the functional entities but this mapping is not identified or required by this specification.

Figure 2-4 shows examples of inheritance hierarchy for Resources. The operations and attributes provided by the *LifeCycle*, *TestableObject*, *PortSupplier*, and *PropertySet* interfaces establish a common approach for interacting with any resource in a SCA environment. The *Port* interface is used for pushing or pulling messages between resources and devices. A resource may consist of zero or more input and output message ports. The figure also shows examples of more specialized resources and devices that result in specific functionality.

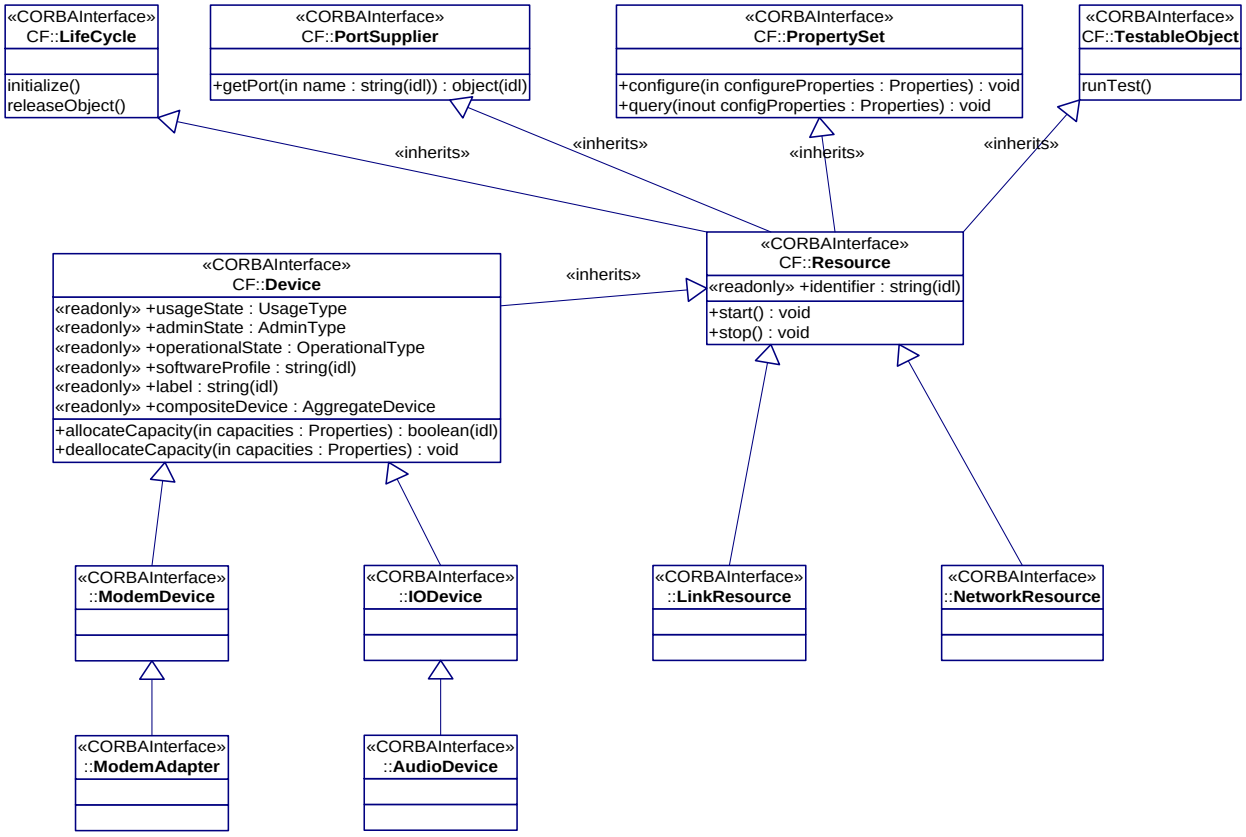

Figure 2-4: Conceptual Model of Resources

2.2.5 Networking Overview

External networking protocols define the communications between a SCA-compliant radio system and its peer systems. A network of nodes is formed between systems which are interconnected by repeaters, bridges, routers, and/or gateways. External-networking protocols will typically communicate peer-to-peer at different layers using physical layer interconnections with a repeater function, link layer interconnections with a bridge function, network layer interconnections with standard network routing, or upper layer interconnections with application gateways.

The different categories of interoperability are outlined below based upon the OSI Model. There may be multiple levels of interoperability within the same system on a waveform-by-waveform basis.

1. **Physical Layer Interoperability.** The external networking protocols provide a compatible physical interface, including the signaling interface, but no higher layer processing. This level of interoperability is adequate for a simple bit-by-bit bridging or relay operation between two interfaces.
2. **Link Layer Interoperability.** The external networking protocols provide link layer processing over all physical interfaces. This level of interoperability is adequate for allowing the radio to be used as transport and for allowing the radio to use another network as transport. Intelligent routing or switching decisions are limited to local layer 2 routing.
3. **Network Layer Interoperability.** The external networking protocols provide network layer address processing interoperability. The radio and the networks being inter-operated are sub-networks of the same Inter-network. At this level, intelligent switching and routing decisions can be made end-to-end.
4. **Host Level Interoperability** (Layers 4 – 7). Embedded applications can exchange information with hosts attached to the network. An example of this is a handheld radio that contains embedded Situation Awareness (SA) application exchanging SA updates with a vehicular platform in an external sub-network. In this example, the radio provides message payload translations to allow two otherwise incompatible hosts to communicate.

In order to support application portability, standard interfaces are required between application protocol entities.

3 SOFTWARE ARCHITECTURE DEFINITION

3.1 OPERATING ENVIRONMENT

This section contains the requirements of the operating system, middleware, 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. An SCA application environment profile (AEP) 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 the SCA AEP is depicted in Figure 3-1. The OE shall provide the functions and options designated as mandatory by the AEP defined in Appendix B. The OE is not limited to providing the functions and options designated as mandatory by the profile. Implementations of the CORBA Object Request Broker (ORB), the CF Framework Control Interfaces, Framework Services Interfaces, and Base Device Interfaces are not limited to using the services designated as mandatory by the SCA AEP.

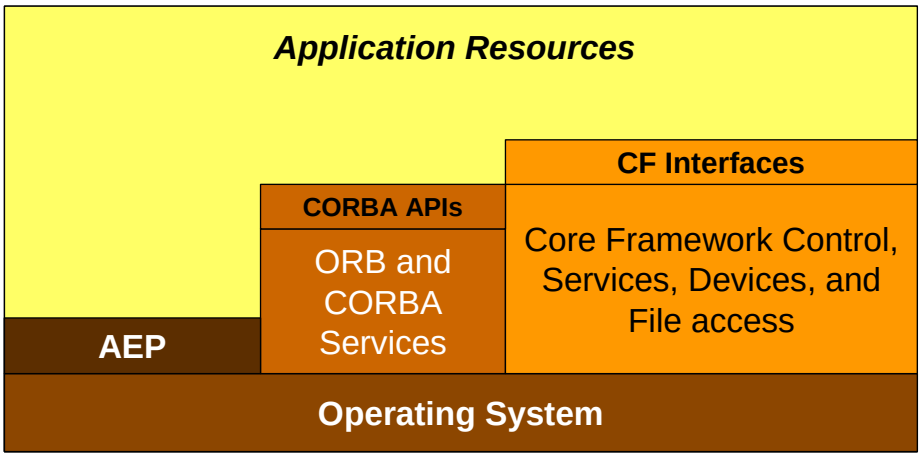

Figure 3-1: Notional Relationship of OE and Application to the SCA AEP

The OE and related file systems shall support a filename length of 40 characters and a 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 3.2)

3.1.2 CORBA Middleware & Services

The OE shall include middleware that, at a minimum, provides the services and capabilities of minimumCORBA as specified by the OMG Document in reference [5].

3.1.2.1 Naming Service

The OE shall provide an implementation of a CORBA Naming Service which implements the CosNaming module *NamingContext* interface operations: *bind*, *bind_new_context*, *unbind*, *destroy*, and *resolve* as defined in the OMG Interoperable Naming Service Specification [6] using the IDL found in Appendix A of that reference.

A Naming Service's NameComponent structure is made up of an id-and-kind pair. The "id" element of each NameComponent is a string value that uniquely identifies a NameComponent. The "kind" element of each NameComponent shall be "" (null string).

3.1.2.2 Log Service

An SCA compliant implementation may include a log service. If a log service is implemented, the log service shall conform to the OMG Lightweight Log Service Specification [7].

3.1.2.2.1 Log Producers

A log producer is a CF component (e.g., *DomainManager*, *Application*, *ApplicationFactory*, *DeviceManager*, *Device*) or an application's CORBA capable component (e.g., *Resource*, *ResourceFactory*) that produces log records using the Lightweight Log Service *CosLwLog::LogProducer* interface. Log records are of type *CosLwLog::ProducerLogRecord.*

Log producers shall implement a configure property which is a CF *Properties* type with an id of "PRODUCER_LOG_LEVEL" and a value that is a *CosLwLog::LogLevelSequence*. The value of this configure property contains all log levels that are enabled. A log producer shall only output log records that contain an enabled *CosLwLog::LogLevel* value. Log levels that are not in the *CosLwLog::LogLevelSequence* are disabled.

Log producers and CF components that are required by this specification to write log records shall operate normally in the absence of a log service or in the case where the connections to a log are nil or an invalid reference.

Log producers shall use their component *identifier* attribute in the producerId field of the *CosLwLog::ProducerLogRecord*.

3.1.2.3 CORBA Event Service and Standard Events

3.1.2.3.1 CORBA Event Service

The OE shall provide an implementation of the CORBA Event Service. The Event Service shall implement the *PushConsumer* and *PushSupplier* interfaces of the CosEventComm module as described in OMG Event Service Specification [8] using the IDL found in that specification.

The CosEventComm CORBA Module is used by consumers for receiving events and by producers for generating events. A component (e.g., *Resource*, *DomainManager*, etc.) that consumes events shall implement the CosEventComm *PushConsumer* interface. A component (e.g., *Resource*, *Device*, *DomainManager*, etc.) that produces events shall implement the CosEventComm *PushSupplier* interface and use the CosEventComm *PushConsumer* interface for generating the events. A producer component shall not forward or raise any exceptions when the connection to a CosEventComm *PushConsumer* is a nil or invalid reference.

The CORBA 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 CORBA objects and communicate with those channels is accomplished using standard CORBA requests. The OE shall provide two standard event channels: Incoming Domain Management and Outgoing Domain Management. The Incoming Domain Management Channel name shall be "IDM_Channel". The Outgoing Domain Management Channel name shall be "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 functions (e.g., *ApplicationFactory*, *Application*, *DomainManager*, etc.). The Outgoing Domain Management Channel is used by domain clients (e.g., HCI) to receive events (e.g., additions or removals from the domain) generated from domain management functions (e.g., *ApplicationFactory*, *Application*, *DomainManager*, etc.). Besides these two standard event channels, the OE allows other event channels to be set up by application developers.

3.1.2.3.2 StandardEvent Module

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

3.1.2.3.2.1 Types

3.1.2.3.2.1.1 StateChangeCategoryType

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

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

3.1.2.3.2.1.2 StateChangeType

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

```
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 */
};
```

3.1.2.3.2.1.3 StateChangeEventType

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

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

3.1.2.3.2.1.4 SourceCategoryType

The type SourceCategoryType is an enumeration that is utilized in the DomainManagementObjectAddedEventType and DomainManagementObjectRemovedEventType. It is used to identify the type of object that has been added to or removed from the domain.

```
enum SourceCategoryType
{
     DEVICE_MANAGER,
     DEVICE,
     APPLICATION_FACTORY,
     APPLICATION,
     SERVICE
};
```

3.1.2.3.2.1.5 DomainManagementObjectRemovedEventType

The type DomainManagementObjectRemovedEventType is a structure used to indicate that the event source has been removed from the domain.

```
struct DomainManagementObjectRemovedEventType
{
     string              producerId;
     string              sourceId;
     string              sourceName;
     SourceCategoryType  sourceCategory;
};
```

3.1.2.3.2.1.6 DomainManagementObjectAddedEventType

The type DomainManagementObjectAddedEventType is a structure used to indicate that the event source has been added to the domain.

```
struct DomainManagementObjectAddedEventType
{
     string              producerId;
     string              sourceId;
     string              sourceName;
```

```
    SourceCategoryType  sourceCategory
    Object              sourceIOR;
};
```

3.1.3 Core Framework

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

Figure 3-2 depicts the key elements of the CF and the IDL relationships between these elements. A *DomainManager* component 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 *Device*, *LoadableDevice*, or *ExecutableDevice* interfaces. Other software components have no direct relationship with a hardware device, but perform application services for the user and implement the *Resource* interface. This interface provides a consistent way of configuring and tearing down these components. Each resource can potentially communicate with other resources. An application is a specific collection of one or more resources which provides a specified service or function and which is managed through the *Application* 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 behavior rules of a resource, and the loading requirements of each resource. The resources may then be created by using the *ResourceFactory* interface or through the *Device* interfaces (*Device*, *LoadableDevice, or ExecutableDevice*) an connected to other resources or devices resident on the system.

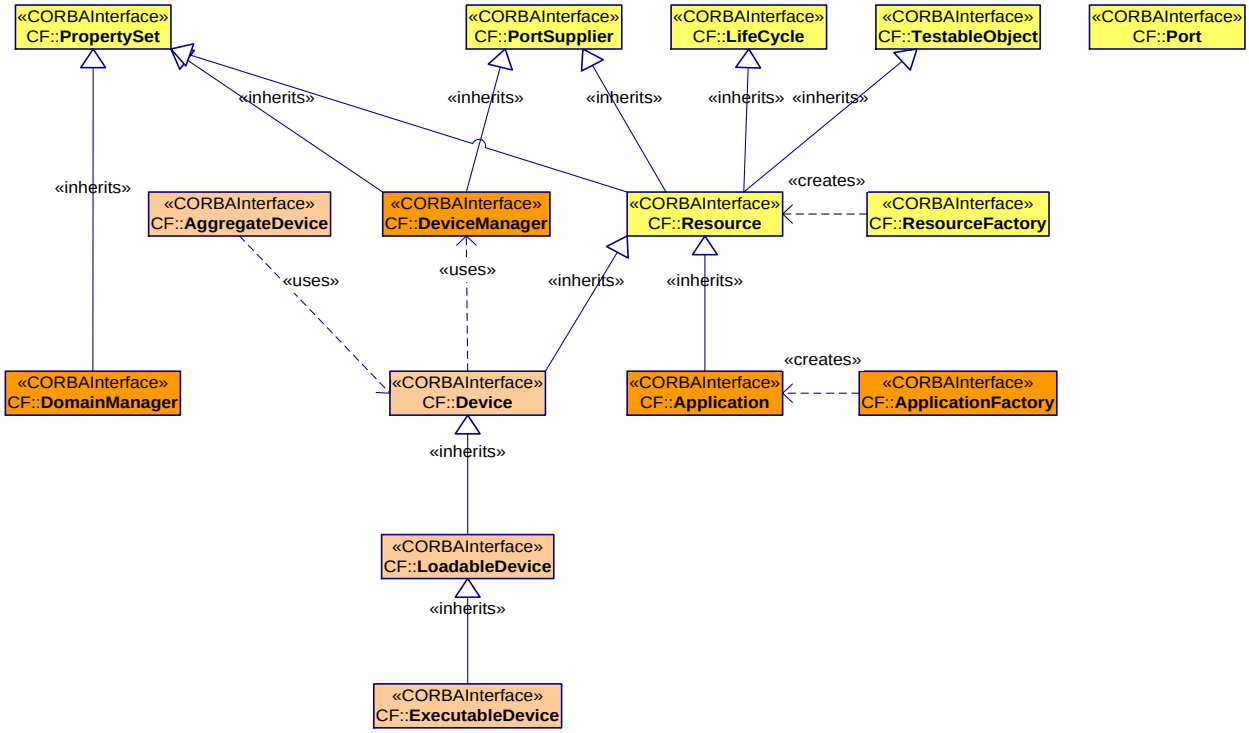

Figure 3-2: Core Framework IDL Relationships

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 Base Application Interfaces

Base Application Interfaces are defined by the Core Framework requirements and implemented by application developers; see section 3.2 for application requirements.

Base Application Interfaces shall be implemented using the CF IDL presented in Appendix C.

3.1.3.1.1 *Port*

3.1.3.1.1.1 Description

This interface provides operations for managing associations between ports. The *Port* interface UML is depicted in Figure 3-3. An application defines a specific port type by specifying an interface that inherits the *Port* interface. 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, stop).

The nature of *Port* fan-in, fan-out, or one-to-one is component dependent.

How components' ports are connected is described in the software assembly descriptor (SAD) and the device configuration descriptor (DCD) files of the Domain Profile (3.1.3.5).

3.1.3.1.1.2 UML

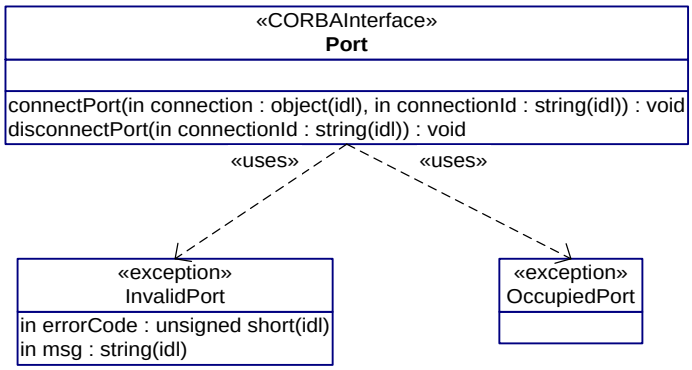

Figure 3-3: *Port* Interface UML

3.1.3.1.1.3 Types

3.1.3.1.1.3.1 InvalidPort

The InvalidPort exception indicates one of the following errors has occurred in the specification of a *Port* association:

1. errorCode 1 means the *Port* component is invalid (unable to narrow object reference) or illegal object reference,
2. errorCode 2 means the *Port* name is not found (not used by this *Port*).

```
exception InvalidPort { unsigned short errorCode; string msg; };
```

3.1.3.1.1.3.2 OccupiedPort

The OccupiedPort exception indicates the port is unable to accept any additional connections.

```
exception OccupiedPort{};
```

3.1.3.1.1.4 Attributes

N/A.

3.1.3.1.1.5 Operations

3.1.3.1.1.5.1 *connectPort*

3.1.3.1.1.5.1.1 Brief Rationale

Applications require the *connectPort* operation to establish associations between ports. Ports provide channels through which data and/or control pass.

The *connectPort* operation provides half of a two-way association; therefore two calls are required to create a two-way association.

3.1.3.1.1.5.1.2 Synopsis

```
void connectPort (in Object connection, in string connectionId)
raises (InvalidPort, OccupiedPort);
```

3.1.3.1.1.5.1.3 Behavior

The *connectPort* operation shall make a connection to the component identified by its input parameters.

A port may support several connections. The input *connectionId* is a unique identifier to be used by the *disconnectPort* operation when breaking a specific connection.

3.1.3.1.1.5.1.4 Returns

This operation does not return a value.

3.1.3.1.1.5.1.5 Exceptions/Errors

The *connectPort* operation shall raise the InvalidPort exception when the input connection parameter is an invalid connection for this port.

The *connectPort* operation shall raise the OccupiedPort exception when unable to accept the connections because the port is already fully occupied.

3.1.3.1.1.5.2 *disconnectPort*

3.1.3.1.1.5.2.1 Brief Rationale

Applications require the *disconnectPort* operation in order to allow consumer/producer data components to disassociate themselves from their counterparts (consumer/producer).

3.1.3.1.1.5.2.2 Synopsis

```
void disconnectPort (in string connectionId) raises
(InvalidPort);
```

3.1.3.1.1.5.2.3 Behavior

The *disconnectPort* operation shall break the connection to the component identified by the input connectionId parameter.

3.1.3.1.1.5.2.4 Returns

This operation does not return a value.

3.1.3.1.1.5.2.5 Exceptions/Errors

The *disconnectPort* operation shall raise the InvalidPort exception when the input connectionId parameter is not a known connection to the *Port* component.

3.1.3.1.2 *LifeCycle*

3.1.3.1.2.1 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 3-4.

3.1.3.1.2.2 UML

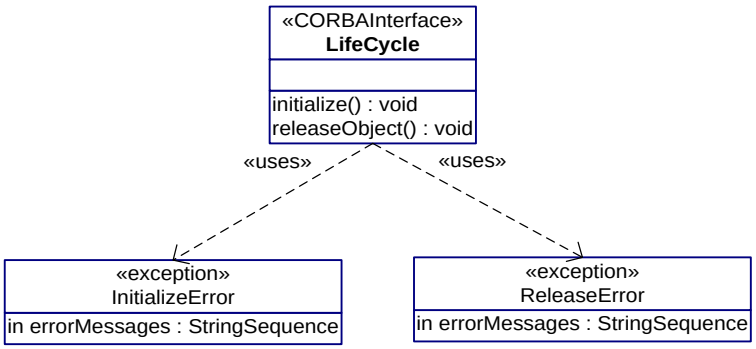

Figure 3-4: *LifeCycle* Interface UML

3.1.3.1.2.3 Types

3.1.3.1.2.3.1 InitializeError

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

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

3.1.3.1.2.3.2 ReleaseError

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

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

3.1.3.1.2.4 Attributes

N/A.

3.1.3.1.2.5 Operations

3.1.3.1.2.5.1 *initialize*

3.1.3.1.2.5.1.1 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, hardware devices may be configured to some state, etc.

3.1.3.1.2.5.1.2 Synopsis

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

3.1.3.1.2.5.1.3 Behavior

Initialization behavior is implementation dependent.

3.1.3.1.2.5.1.4 Returns

This operation does not return a value.

3.1.3.1.2.5.1.5 Exceptions/Errors

The *initialize* operation shall raise an InitializeError exception when an initialization error occurs.

3.1.3.1.2.5.2 *releaseObject*

3.1.3.1.2.5.2.1 Brief Rationale

The purpose of the *releaseObject* operation is to provide a means by which an instantiated component may be torn down.

3.1.3.1.2.5.2.2 Synopsis

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

3.1.3.1.2.5.2.3 Behavior

The *releaseObject* operation shall release all internal memory allocated by the component during the life of the component. The *releaseObject* operation shall tear down the component and release it from the CORBA environment.

3.1.3.1.2.5.2.4 Returns

This operation does not return a value.

3.1.3.1.2.5.2.5 Exceptions/Errors

The *releaseObject* operation shall raise a ReleaseError exception when a release error occurs.

3.1.3.1.3 *TestableObject*

3.1.3.1.3.1 Description

The *TestableObject* interface defines a set of operations that is used to test component implementations. The *TestableObject* interface UML is depicted in Figure 3-5.

3.1.3.1.3.2 UML

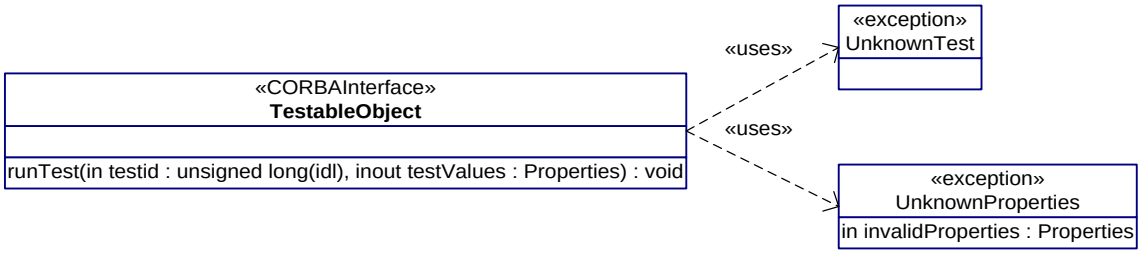

Figure 3-5: *TestableObject* Interface UML

3.1.3.1.3.3 Types

3.1.3.1.3.3.1 UnknownTest

The UnknownTest exception indicates the input testId parameter is not known by the component.

```
exception UnknownTest{};
```

3.1.3.1.3.4 Attributes

N/A.

3.1.3.1.3.5 Operations

3.1.3.1.3.5.1 *runTest*

3.1.3.1.3.5.1.1 Brief Rationale

The *runTest* operation allows components to be "black box" tested. This allows built-in tests (BITs) to be implemented which provide a means to isolate faults (both software and hardware) within the system.

3.1.3.1.3.5.1.2 Synopsis

```
void runTest (in unsigned long testId, inout Properties
testValues) raises (UnknownTest, UnknownProperties);
```

3.1.3.1.3.5.1.3 Behavior

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 shall be used to provide additional information to the implementation-specific test to be run. 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 Properties Descriptor. Valid testId(s) and both input and output testValues (properties) for the *runTest* operation shall at a minimum be the test properties defined in the *properties test* element of the component's Properties Descriptor (refer to Appendix D Domain Profile). The testId parameter corresponds to the XML attribute *testId* of the property element *test* in a propertyfile.

A CF UnknownProperties exception is raised by the *runTest* operation. All testValues parameter properties (i.e., *test* properties defined in the propertyfile(s) referenced in the component's SPD) shall be validated.

The *runTest* operation shall 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.

3.1.3.1.3.5.1.4 Returns

This operation does not return a value.

3.1.3.1.3.5.1.5 Exceptions/Errors

The *runTest* operation shall raise the UnknownTest exception when there is no underlying test implementation that is associated with the input testId given.

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. 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.

3.1.3.1.4 *PortSupplier*

3.1.3.1.4.1 Description

This interface provides the *getPort* operation for those components that provide ports.

3.1.3.1.4.2 UML

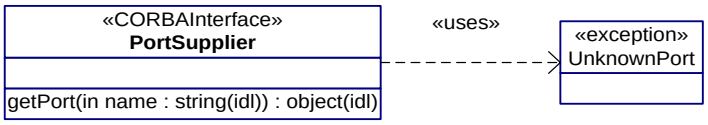

Figure 3-6: *PortSupplier* Interface UML

3.1.3.1.4.3 Types

3.1.3.1.4.3.1 UnknownPort

The UnknownPort exception is raised if an undefined port is requested.

```
exception UnknownPort{};
```

3.1.3.1.4.4 Attributes

N/A.

3.1.3.1.4.5 Operations

3.1.3.1.4.5.1 *getPort*

3.1.3.1.4.5.1.1 Brief Rationale

The *getPort* operation provides a mechanism to obtain a specific consumer or producer port. A port supplier may contain zero-to-many consumer and producer port components. The exact number is specified in the component's software profile SCD (section 3.1.3.5). Multiple input and/or output ports provide flexibility for port suppliers that manage varying priority levels and categories of incoming and outgoing messages, provide multi-threaded message handling, or other special message processing.

3.1.3.1.4.5.1.2 Synopsis

```
Object getPort (in string name) raises (UnknownPort);
```

3.1.3.1.4.5.1.3 Behavior

The *getPort* operation returns the object reference to the named port as stated in the component's SCD.

3.1.3.1.4.5.1.4 Returns

The *getPort* operation shall return the CORBA object reference that is associated with the input port name.

3.1.3.1.4.5.1.5 *Exceptions*/Errors

The *getPort* operation shall raise an UnknownPort exception if the port name is invalid.

3.1.3.1.5 *PropertySet*

3.1.3.1.5.1 Description

The *PropertySet* interface defines *configure* and *query* operations to access component properties/attributes. The *PropertySet* interface UML is depicted in Figure 3-7.

3.1.3.1.5.2 UML

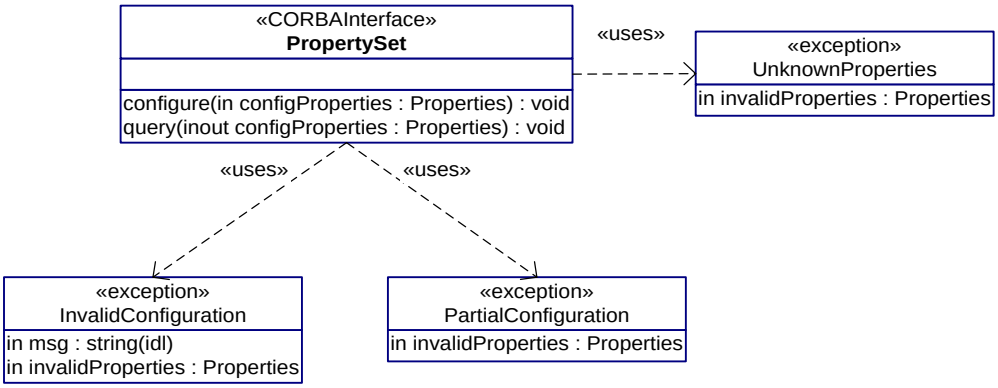

Figure 3-7: *PropertySet* Interface UML

3.1.3.1.5.3 Types

N/A.

3.1.3.1.5.3.1 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.

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

3.1.3.1.5.3.2 PartialConfiguration

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

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

3.1.3.1.5.4 Attributes

N/A.

3.1.3.1.5.5 Operations

3.1.3.1.5.5.1 *configure*

3.1.3.1.5.5.1.1 Brief Rationale

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

3.1.3.1.5.5.1.2 Synopsis

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

3.1.3.1.5.5.1.3 Behavior

The *configure* operation shall assign values to the properties as indicated in the input configProperties parameter. Valid properties for the *configure* operation shall at a minimum be the configure readwrite and writeonly properties referenced in the component's SPD.

3.1.3.1.5.5.1.4 Returns

This operation does not return a value.

3.1.3.1.5.5.1.5 Exceptions/Errors

The *configure* operation shall raise a PartialConfiguration exception when some configuration properties were successfully set and some configuration properties were not successfully set.

The *configure* operation shall raise an InvalidConfiguration exception when a configuration error occurs and no configuration properties were successfully set.

3.1.3.1.5.5.2 *query*

3.1.3.1.5.5.2.1 Brief Rationale

The *query* operation allows a component to be queried to retrieve its properties.

3.1.3.1.5.5.2.2 Synopsis

```
void query (inout Properties configProperties) raises
(UnknownProperties);
```

3.1.3.1.5.5.2.3 Behavior

The *query* operation shall return all component properties when the inout parameter configProperties is zero size. The *query* operation shall return only those id/value pairs specified in the configProperties parameter if the parameter is not zero size. Valid properties for the *query* operation shall be all *configure* properties (simple properties whose *kind* element's *kindtype* attribute is "configure") whose *mode* attribute is "readwrite" or "readonly" and any *allocation* properties with an *action* value of "external" as referenced in the component's SPD.

3.1.3.1.5.5.2.4 Returns

This operation does not return a value.

3.1.3.1.5.5.2.5 Exceptions/Errors

The *query* operation shall raise the CF UnknownProperties exception when one or more properties being requested are not known by the component.

3.1.3.1.6 *Resource*

3.1.3.1.6.1 Description

The *Resource* interface provides a common API for the control and configuration of a software component. The *Resource* interface UML is depicted in Figure 3-8.

The Resource interface inherits from the *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interfaces.

The inherited *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interface operations are documented in their respective sections of this document.

The *Resource* interface may also be inherited by other application interfaces as described in the software profile's Software Component Descriptor (SCD) file (see 3.1.3.5.2).

3.1.3.1.6.2 UML.

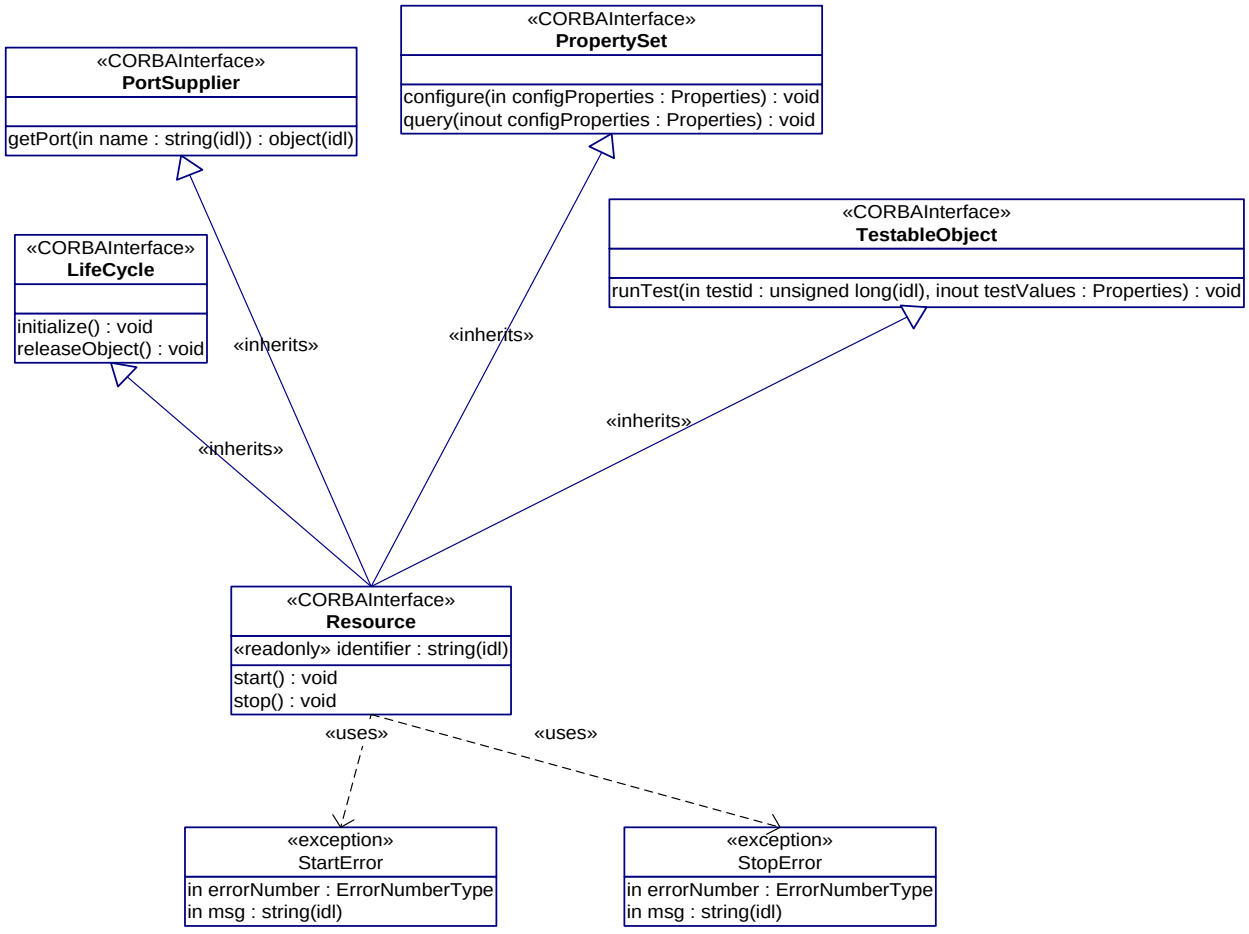

Figure 3-8: Resource Interface UML

3.1.3.1.6.3 Types

3.1.3.1.6.3.1 StartError

The StartError exception indicates that an error occurred during an attempt to start the resource. The errorNumber parameter shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.1.6.3.2 StopError

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

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

3.1.3.1.6.4 Attributes

3.1.3.1.6.4.1 identifier

The readonly identifier attribute shall contain the unique identifier for a *Resource* instance.

```
readonly attribute string identifier;
```

3.1.3.1.6.5 Operations

3.1.3.1.6.5.1 *start*

3.1.3.1.6.5.1.1 Brief Rationale

The *start* operation is provided to command the resource implementing this interface to start internal processing.

3.1.3.1.6.5.1.2 Synopsis

```
void start()raises (StartError);
```

3.1.3.1.6.5.1.3 Behavior

The *start* operation puts the resource in an operating condition.

3.1.3.1.6.5.1.4 Returns

This operation does not return a value.

3.1.3.1.6.5.1.5 Exceptions/Errors

The *start* operation shall raise the StartError exception if an error occurs while starting the resource.

3.1.3.1.6.5.2 *stop*

3.1.3.1.6.5.2.1 Brief Rationale

The *stop* operation is provided to command the resource implementing this interface to stop internal processing.

3.1.3.1.6.5.2.2 Synopsis

```
void stop()raises (StopError);
```

3.1.3.1.6.5.2.3 Behavior

The *stop* operation shall disable all current operations and put the resource in a non-operating condition. The *stop* operation shall not inhibit subsequent *configure*, *query*, and *start* operations.

3.1.3.1.6.5.2.4 Returns

This operation does not return a value.

3.1.3.1.6.5.2.5 Exceptions/Errors

The *stop* operation shall raise the StopError exception if an error occurs while stopping the resource.

3.1.3.1.7 *ResourceFactory*

3.1.3.1.7.1 Description

A resource factory is used to create and tear down a resource. The *ResourceFactory* interface is designed after the Factory Design Patterns. The *ResourceFactory* interface UML is depicted in Figure 3-9. The factory mechanism provides client-server isolation among resources and provides a standard mechanism of obtaining a resource without knowing its identity. An application is not required to use resource factories to obtain, create, or tear down resources. A software profile specifies which application resource factories are to be used by the application factory.

3.1.3.1.7.2 UML

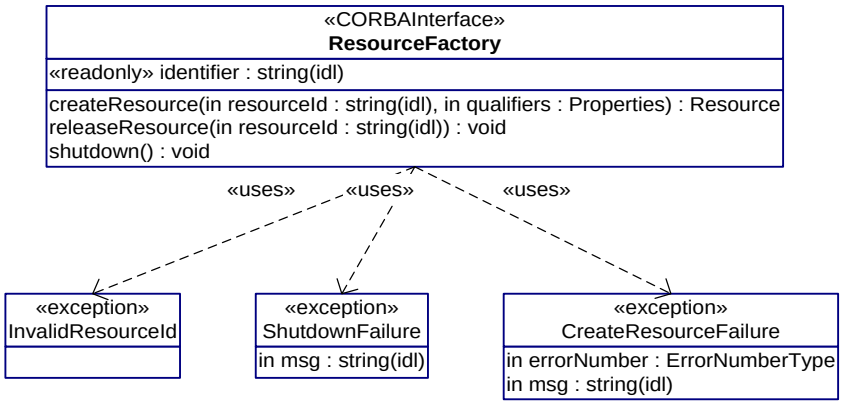

Figure 3-9: *ResourceFactory* Interface UML

3.1.3.1.7.3 Types

3.1.3.1.7.3.1 InvalidResourceId

The InvalidResourceId exception indicates the resourceId does not reference a resource created by this resource factory.

```
exception InvalidResourceId{};
```

3.1.3.1.7.3.2 ShutdownFailure

The ShutdownFailure exception indicates that the *shutdown* method failed to release the resource factory from the CORBA environment. The message is component-dependent, providing additional information describing why the shutdown failed.

```
exception ShutdownFailure { string msg; };
```

3.1.3.1.7.3.3 CreateResourceFailure

The CreateResourceFailure exception indicates that the *createResource* operation failed to create the resource. The error number shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.1.7.4 Attributes

3.1.3.1.7.4.1 identifier

The readonly identifier attribute shall contain the unique identifier for a *ResourceFactory* instance.

```
readonly attribute string identifier;
```

3.1.3.1.7.5 Operations

3.1.3.1.7.5.1 *createResource*

3.1.3.1.7.5.1.1 Brief Rationale

The *createResource* operation provides the capability to create resources in the same process space as the resource factory or to return a reference to a resource that has already been created. This behavior is an alternative approach to the *Device::execute* operation for creating a resource.

3.1.3.1.7.5.1.2 Synopsis

```
Resource createResource (in string resourceId, in Properties
qualifiers) raises (CreateResourceFailure);
```

3.1.3.1.7.5.1.3 Behavior

The resourceId parameter is the identifier for a resource. The qualifiers parameter contains values used by the resource factory in creation of the Resource. The application factory determines the values to be supplied for the qualifiers from the description in the resource factory's software profile. The qualifiers may be used to identify, for example, specific subtypes of resources created by a resource factory.

The *createResource* operation shall create a resource if no resource exists for the given *resourceId* and shall assign the given resourceId to a new resource. If the resource already exists for the given resourceId, the input qualifiers parameter is ignored and the resource's reference is returned. The *createResource* operation shall set a reference count to one, when the resource is initially created, or increment the reference count by one, when the resource already exists. The reference count is used to indicate the number of times that a specific resource reference has been given to requesting clients. This ensures that the resource factory does not release a resource that has a reference count greater than zero (0). When multiple clients have obtained a

reference to the same resource, each client requests release of the resource when it is through with the resource. However, the resource is not released until the release request comes from the last client in existence.

3.1.3.1.7.5.1.4 Returns

The *createResource* operation shall return a reference to the created resource. If the resource already exists, the *createResource* operation shall return a reference to the existing resource.

3.1.3.1.7.5.1.5 Exceptions/Errors

The *createResource* operation shall raise the CreateResourceFailure exception when it cannot create the resource.

3.1.3.1.7.5.2 *releaseResource*

3.1.3.1.7.5.2.1 Brief Rationale

In CORBA there is client side and server side representation of a resource. The *releaseResource* operation provides the mechanism of releasing the resource in the CORBA environment on the server side when all clients are through with a specific resource. The client still has to release its client side reference of the resource.

3.1.3.1.7.5.2.2 Synopsis

```
void releaseResource (in string resourceId) raises
{InvalidResourceId);
```

3.1.3.1.7.5.2.3 Behavior

The *releaseResource* operation shall decrement the reference count for the specified resource, as indicated by the resourceId parameter. The *releaseResource* operation shall release the resource from the CORBA environment and make the resource no longer available when the resource's reference count is zero.

3.1.3.1.7.5.2.4 Returns

This operation does not return a value.

3.1.3.1.7.5.2.5 Exceptions/Errors

The *releaseResource* operation shall raise the InvalidResourceId exception if an invalid resourceId is received.

3.1.3.1.7.5.3 *shutdown*

3.1.3.1.7.5.3.1 Brief Rationale

In CORBA there is client side and server side representation of a resource factory. The *shutdown* operation provides the mechanism for releasing the resource factory from the CORBA environment on the server side. The client has the responsibility to release its client side reference of the resource factory.

3.1.3.1.7.5.3.2 Synopsis

```
void shutdown() raises {ShutdownFailure);
```

3.1.3.1.7.5.3.3 Behavior

The *shutdown* operation shall release the resource factory from the CORBA environment and make it unavailable to any subsequent calls to its object reference.

3.1.3.1.7.5.3.4 Returns

This operation does not return a value.

3.1.3.1.7.5.3.5 Exceptions/Errors

The *shutdown* operation shall raise the ShutdownFailure exception when processing errors prevent the release of the resource factory from the CORBA environment or when all resources have not been released from the resource factory.

3.1.3.2 Framework Control Interfaces

Framework control within a Domain is accomplished by domain management and device management interfaces.

The management interfaces are *Application*, *ApplicationFactory*, *DeviceManager* and *DomainManager*. These interfaces 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 *Application*, *ApplicationFactory*, and *DomainManager* interfaces are coupled together and are delivered together as a complete domain management implementation and service.

Device management is accomplished by the *DeviceManager* interface. The device manager is responsible for creation of logical devices and launching service applications on these logical devices.

Framework Control Interfaces shall be implemented using the CF IDL presented in Appendix C.

3.1.3.2.1 *Application*

3.1.3.2.1.1 Description

The *Application* class provides the interface for the control, configuration, and status of an instantiated application in the domain.

The *Application* interface inherits the IDL interface of *Resource*. A created application instance may contain *Resource* components and/or non-CORBA components. The *Application* interface UML is depicted in Figure 3-10.

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

An instance of an *Application* is returned by the *create* operation of an instance of the *ApplicationFactory* class.

3.1.3.2.1.2 UML

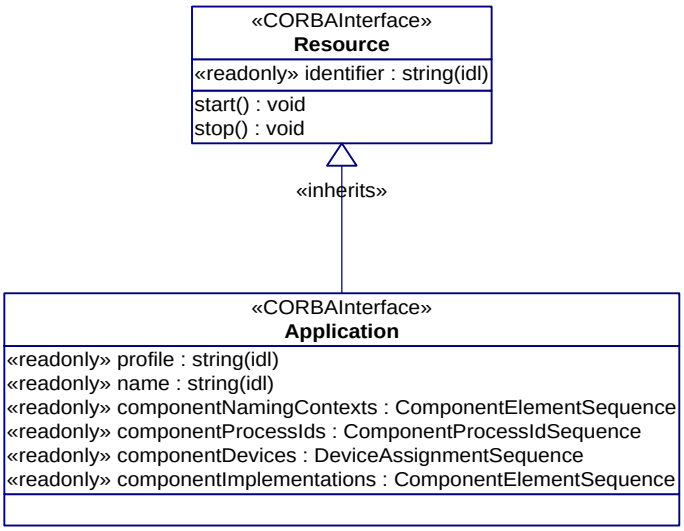

Figure 3-10: *Application* Interface UML

3.1.3.2.1.3 Types

3.1.3.2.1.3.1 ComponentProcessIdType

The ComponentProcessIdType defines a type for associating a component with its process ID.

```
struct ComponentProcessIdType
{
     string componentId;
     unsigned long processId;
};
```

3.1.3.2.1.3.2 ComponentProcessIdSequence

The ComponentProcessIdSequence type defines an unbounded sequence of components' process IDs.

```
typedef sequence  <ComponentProcessIdType>
ComponentProcessIdSequence;
```

3.1.3.2.1.3.3 ComponentElementType

The ComponentElementType defines a type for associating a component with an element (e.g., naming context, implementation ID).

```
struct ComponentElementType
{
     string componentId;
     string elementId;
};
```

3.1.3.2.1.3.4 ComponentElementSequence

The ComponentElementSequence defines an unbounded sequence of ComponentElementType.

```
typedef sequence  <ComponentElementType>
ComponentElementSequence;
```

3.1.3.2.1.4 Attributes

3.1.3.2.1.4.1 profile

The readonly profile attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the application's SAD file. Files referenced within the profile are obtained via a *FileManager*.

```
readonly attribute string profile;
```

3.1.3.2.1.4.2 name

This readonly name attribute shall contain the name of the created application. The *ApplicationFactory* interface's *create* operation name parameter provides the name content.

```
readonly attribute string name;
```

3.1.3.2.1.4.3 componentNamingContexts

The componentNamingContexts attribute shall contain the list of components' Naming Service Context within the application for those components using CORBA Naming Service.

```
readonly attribute ComponentElementSequence
componentNamingContexts;
```

3.1.3.2.1.4.4 componentProcessIds

The componentProcessIds attribute shall contain the list of components' process IDs within the Application for components that are executing on a device.

```
readonly attribute ComponentProcessIdSequence
componentProcessIds;
```

3.1.3.2.1.4.5 componentDevices

The componentDevices attribute shall contain a list of devices, which each component either uses, is loaded on or is executed on. Each component (identified by the *componentinstantiation* element in the application's software profile) is associated with at least one device.

```
readonly attribute DeviceAssignmentSequence componentDevices;
```

3.1.3.2.1.4.6 componentImplementations

The componentImplementations attribute shall contain the list of components' SPD implementation IDs within the application for those components created.

```
readonly attribute ComponentElementSequence
componentImplementations;
```

3.1.3.2.1.5 General Class Behavior

The application shall delegate the implementation of the inherited *Resource* operations (*runTest*, *start*, *stop*, *configure*, and *query*) to the *Application Resource* component identified by the application's SAD *assemblycontroller* element (Assembly Controller). The application shall propagate exceptions raised by the application's Assembly Controller's operations. The

initialize operation shall not be propagated to the application's components or its Assembly Controller.

3.1.3.2.1.6 Operations

3.1.3.2.1.6.1 *releaseObject*

3.1.3.2.1.6.1.1 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 terminating, the application removes the message connectivity with its associated applications (e.g., ports, resources, and logs) in the domain.

3.1.3.2.1.6.1.2 Synopsis

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

3.1.3.2.1.6.1.3 Behavior

The following behavior is in addition to the *LifeCycle*::*releaseObject* operation behavior.

The *Application::releaseObject* operation shall release each application component not created by a resource factory by utilizing the component's *Resource::releaseObject* operation. The *Application::releaseObject* operation shall release each component created by a resource factory via the *ResourceFactory::releaseResource* operation. The *Application::releaseObject* operation shall terminate a resource factory when no more resources are managed by the resource factory via the *ResourceFactory::shutdown* operation. The *Application::releaseObject* operation shall terminate the processes / tasks on allocated executable devices belonging to each application component by utilizing the *ExecutableDevice:terminate* operation.

The *releaseObject* operation shall de-allocate the memory associated with each application component instance from its allocated device by utilizing the *LoadableDevice::unload* operation.

The *releaseObject* operation shall deallocate the device capacities that were allocated during application creation. The actual devices deallocated (*Device::deallocateCapacity*) reflect changes in their capacity based upon component capacity requirements deallocated from them, which may also cause state changes for the devices.

The application shall release all object references to the components making up the application.

The *releaseObject* operation shall disconnect ports that were previously connected based upon the application's software profile.

The *releaseObject* operation shall disconnect consumers and producers from a CORBA Event Service's event channel based upon the software profile. The *releaseObject* operation may destroy a CORBA Event Service's event channel when no more consumers and producers are connected to it.

For components (e.g., *Resource*, *ResourceFactory*) that are registered with Naming Service, the *releaseObject* operation shall unbind those components and destroy the associated naming contexts as necessary from the Naming Service.

The *releaseObject* operation for an application shall disconnect ports first, then release the resources and the resource factories, then call the *terminate* operation, and lastly call the *unload* operation on the devices.

The *releaseObject* operation shall, upon successful application release, write an ADMINISTRATIVE_EVENT log record.

The *releaseObject* operation shall, upon unsuccessful application release, write a FAILURE_ALARM log record.

The *releaseObject* operation shall send a DomainManagementObjectRemovedEventType 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 released application.
2. The *sourceId* is the identifier attribute of the released application.
3. The *sourceName* is the name attribute of the released application.
4. The *sourceCategory* is "APPLICATION".

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

1. Client invokes *releaseObject* operation.
2. Disconnect *Ports*.
3. Release the *ResourceFactory* components.
4. Shutdown the *ResourceFactory* components.
5. Release the *Resource* components.
6. Terminate the components' processes.
7. Unload the components' executable images.
8. Change the state of the associated devices to be available, along with device(s) memory utilization availability and processor utilization availability based upon the Device Profile and software profile.
9. Unbind application components from Naming Service.
10. Log an Event indicating that the application was either successfully or unsuccessfully released.
11. Remove the application reference from the applications attribute and generate an event to indicate the application has been removed from the domain.

Figure 3-11 is a collaboration diagram depicting the behavior as described above.

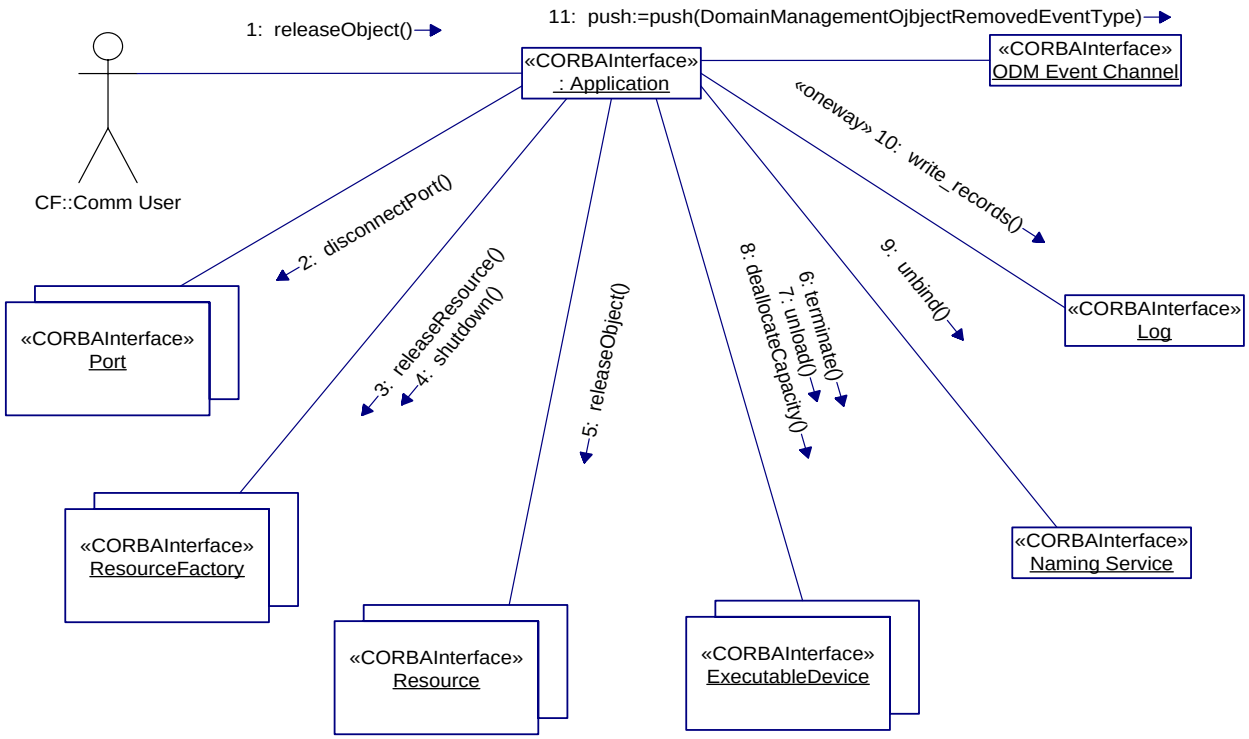

Figure 3-11: *Application* Behavior

3.1.3.2.1.6.1.4 Returns

This operation does not return a value.

3.1.3.2.1.6.1.5 Exceptions/Errors

The *releaseObject* operation shall raise a *ReleaseError* exception when internal processing errors prevent the successful release of any application component.

3.1.3.2.1.6.2 *getPort*

3.1.3.2.1.6.2.1 Brief Rationale

The *getPort* operation obtains an object reference to a specific visible port of the application.

3.1.3.2.1.6.2.2 Synopsis

```
Object getPort (in string name) raises (UnknownPort);
```

3.1.3.2.1.6.2.3 Behavior

The *getPort* operation returns object references for port names that are in the application SAD *externalports* element.

3.1.3.2.1.6.2.4 Returns

The *getPort* operation shall return object references only for input port names that match the port names that are in the application SAD *externalports* element.

3.1.3.2.1.6.2.5 Exceptions/Errors

The *getPort* operation shall raise an UnknownPort exception if the port is invalid.

3.1.3.2.2 *ApplicationFactory*

3.1.3.2.2.1 Description

The *ApplicationFactory* interface class provides an interface to request the creation of a specific type of application in the domain.

The *ApplicationFactory* interface class is designed using the Factory Design Pattern. The software profile determines the type of application that is created by the application factory.

3.1.3.2.2.2 UML

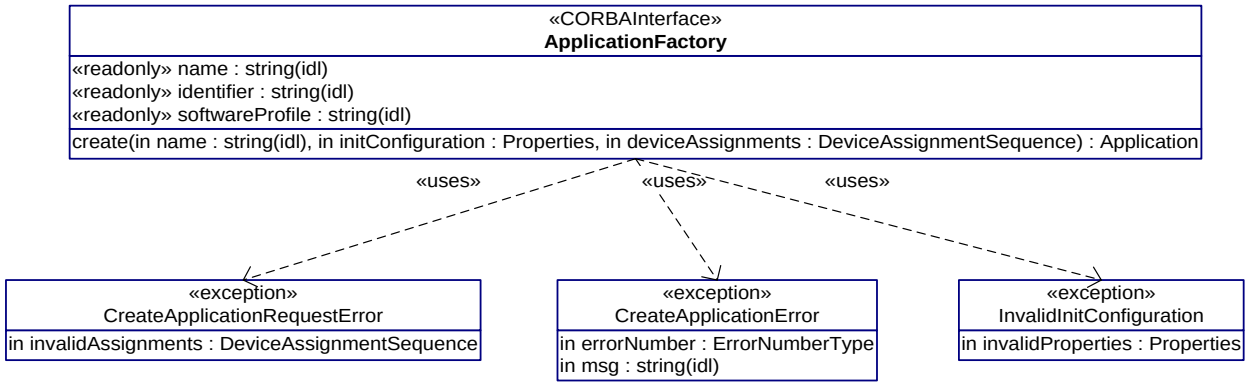

Figure 3-12: *ApplicationFactory* UML

3.1.3.2.2.3 Types

3.1.3.2.2.3.1 CreateApplicationRequestError Exception

The CreateApplicationRequestError exception is raised when the parameter CF DeviceAssignmentSequence contains one (1) or more invalid application component-to-device assignment(s).

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

3.1.3.2.2.3.2 CreateApplicationError Exception

The CreateApplicationError exception is raised when a *create* request is valid but the application is unsuccessfully instantiated due to internal processing errors. The error number shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.2.2.3.3 Exception InvalidInitConfiguration

The InvalidInitConfiguration exception is raised when the input initConfiguration parameter is invalid.

```
exception InvalidInitConfiguration { Properties
invalidProperties; };
```

3.1.3.2.2.4 Attributes

3.1.3.2.2.4.1 name

The readonly name attribute contains the user-friendly name of the application instantiated by an application factory. The name attribute shall be identical to the *softwareassembly* element *name* attribute of the application's Software Assembly Descriptor file.

```
readonly attribute string name;
```

3.1.3.2.2.4.2 softwareProfile

The softwareProfile attribute contains the Profile Descriptor for the application that is created by the application factory.

The readonly softwareProfile attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the application's SAD file. Files referenced within the profile are obtained via *FileManager*.

```
readonly attribute string softwareProfile;
```

3.1.3.2.2.4.3 identifier

The readonly identifier attribute shall contain the unique identifier for an *ApplicationFactory* instance. The identifier shall be identical to the *softwareassembly* element *id* attribute of the application factory's Software Assembly Descriptor file.

```
readonly attribute string identifier;
```

3.1.3.2.2.5 Operations

3.1.3.2.2.5.1 *create*

3.1.3.2.2.5.1.1 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.

3.1.3.2.2.5.1.2 Synopsis

```
Application create (in string name, in Properties
initConfiguration, in DeviceAssignmentSequence
deviceAssignments) raises (CreateApplicationError,
CreateApplicationRequestError, InvalidInitConfiguration);
```

3.1.3.2.2.5.1.3 Behavior

The *create* operation shall use the SPD *implementation* element to locate candidate devices capable of loading and executing application software modules.

The *create* operation validates all component-device associations in the input deviceAssignments parameter by verifying that the device indicated by the assignedDeviceId element provides the

necessary capacities and properties required by the component indicated by the componentId element. Device assignments should not be given for resources created via a resource factory since instantiation of these *Resources* is controlled by the creating *ResourceFactory*.

The *create* operation shall perform the comparison of allocation properties of the application to those of each candidate device, according to the allocation property's *action* element, for those application component properties whose *kindtype* is *allocation* and whose *action* element is not *external*.

The *create* operation shall use the *allocateCapacity* operation to perform the comparison of allocation properties of the application to those of each candidate device for those application component properties whose *kindtype* is *allocation* and whose *action* element is *external*

The *create* operation shall deallocate any capacity allocations on devices that do not satisfy the application components allocation requirements or that are not utilized due to an unsuccessful application creation.

The *create* operation shall load application modules onto devices that have been granted successful capacity allocations and that satisfy the application components allocation requirements.

The *create* operation shall execute the application software modules as specified in the application's Software Assembly Descriptor (SAD) file. The *create* operation shall use each software module's SPD implementation code's stack size and priority elements, when specified, for the execute options parameters.

The *create* operation shall include the mandatory execute parameters Naming Context IOR, Name Binding, and Component Identifier, as described in this section, in the parameters parameter of the *ExecutableDevice::execute* operation when the CORBA instance's *componentinstantiation* element of the SAD contains a *findcomponent* element with a *namingservice* sub-element.

The execute parameter for the Naming Context IOR shall be a CF *Properties* type with an id element set to "NAMING_CONTEXT_IOR" and a value element set to the stringified IOR of the naming context to which the component will bind. The *create* operation shall create any naming contexts that do not exist but which are required for successful binding to the Naming Context IOR. The structure of the naming context path shall be "/ DomainName / [optional naming context sequences]". In the naming context path, each "slash" (/) represents a separate naming context.

The Name Binding execute parameter shall be a CF *Properties* type with an id element set to "NAME_BINDING" and a value element set to a string in the format of "ComponentName_UniqueIdentifier". The ComponentName value is the SAD *componentinstantiation findcomponent namingservice* element's *name* attribute. The UniqueIdentifier is determined by the implementation. The Name Binding parameter is used by the component to bind its object reference to the Naming Context IOR parameter.

The Component Identifier execute parameter shall be 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 shall be identical to the *create* operation's input name parameter. The Application_Name field provides a specific instance qualifier for executed components.

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. The *create* operation passes "execparam" parameters values as string values.

Upon execution of a software module by the *create* operation, a *Resource* or a *ResourceFactory* component shall register with the Naming Service. The *create* operation uses "ComponentName_UniqueIdentifier" to retrieve the component's CORBA object reference from the Naming Context IOR.

The *create* operation obtains a resource in accordance with the SAD via the CORBA Naming Service or a resource factory. The *ResourceFactory* object reference is obtained by using the CORBA Naming Service. The *create* operation, when creating a resource from a resource factory, shall pass the *componentinstantiation componentresourcefactoryref* element properties whose *kindtype* element is *factoryparam* as the *qualifiers* parameter to the referenced *ResourceFactory* component's *createResource* operation.

The *create* operation shall, in order, initialize all application resources, then establish connections for those resources, and finally configure the application component indicated by the *assemblycontroller* element in the SAD. The *create* operation connects the ports of the application resources with the ports of other resources within the application as well as the devices and services they use in accordance with the SAD.

The *create* operation shall establish connections for an application which are specified in the SAD *domainfinder* element. The *create* operation obtains object references to the required *Port* interfaces in via *PortSupplier::getPort* operation. The *create* operation uses the SAD *connectinterface* element *id* attribute as the unique identifier for a specific connection when provided. The *create* operation creates a connection id when no SAD *connectinterface* element *id* attribute is specified for a connection. For connections to an event channel, the *create* operation shall connect a *CosEventComm::PushConsumer* or *CosEventComm::PushSupplier* object to the event channel as specified in the SAD's *domainfinder* element. The *create* operation shall create the specified event channel if the event channel does not exist.

The *create* operation shall configure the application component indicated by the *assemblycontroller* element in the SAD if that component has properties with a *kindtype* of "configure" and a *mode* of "readwrite" or "writeonly". The *create* operation shall use the union of the properties contained in the input initConfiguration parameter of the *create* operation and the assembly controller's *componentinstantiation* element properties with a *kindtype* of "configure" and a *mode* of "readwrite" or "writeonly". Values contained in the input initConfiguration parameter shall have precedence over the values of the assembly controller's *componentinstantiation* element properties when they reference the same property.

The *TestableObject*::*runTest* operation (3.1.3.1.3.5.1), *Resource*::*stop* operation (3.1.3.1.6.5.1), and *Resource*::*start* operation (3.1.3.1.6.5.1) are not called at start-up.

The *create* operation shall return an *Application* object reference for the created application when the application is successfully created.

The *create* operation shall, upon successful application creation, write an ADMINISTRATIVE_EVENT log record.

The *create* operation shall, upon unsuccessful application creation, write a FAILURE_ALARM log record.

The *create* operation shall send a DomainManagementObjectAddedEventType event to the Outgoing Domain Management event channel upon successful creation of an application. For this event:

1. The *producerId* is the identifier attribute of the application factory.
2. The *sourceId* is the identifier attribute of the created application.
3. The *sourceName* is the name attribute of the created application.
4. The *sourceIOR* is the object reference for the created application.
5. The *sourceCategory* is "APPLICATION".

The following steps demonstrate one scenario of the behavior of an application factory for the creation of an application:

1. Client invokes the *create* operation.
2. 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. Create an instance of an *Application*, if the requested application can be created. Update the memory and processor utilization of the devices.
3. Allocate the device(s) memory and processor utilization.
4. Load the application software modules on the devices using the appropriate *Device*(s) interface provided the application software modules haven't already been loaded.
5. Execute the application software modules on the devices using the appropriate *Device* interface as indicated by the application's software profile.
6. Obtain the object reference (*Resource* or *ResourceFactory*) as described by the SAD.
7. If the component obtained from the CORBA Naming Service is a resource factory as indicated by the SAD, then narrow the object reference to be a *ResourceFactory* component.
8. If the component is a *ResourceFactory*, then create a resource using the *ResourceFactory* interface.
9. If the components obtained from the Naming Services is a resource supporting the *Resource* interface as indicated by the SCDs, then narrow the components reference to *Resource* components.
10. Initialize the resource.
11. Get *Port* object references for the resources.

12. Connect the ports that interconnect the resources' ports together.
13. Configure the *assemblycontroller* component using the *Resource* interface.
14. Return the *Application* object reference and log message.
15. Generate an event to indicate the application has been added to the domain.

Figure 3-13 is a collaboration diagram depicting the behavior as described above.

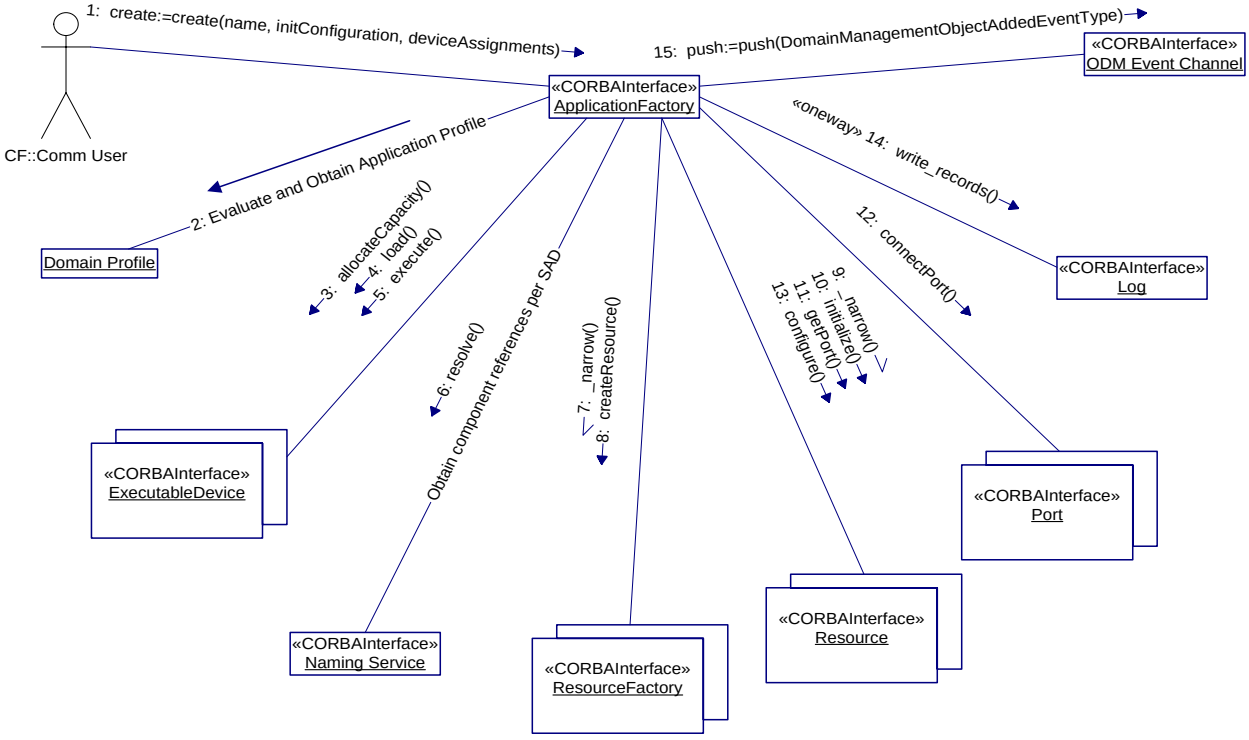

Figure 3-13: *ApplicationFactory* Behavior

3.1.3.2.2.5.1.4 Returns

The *create* operation returns a duplicated *Application* reference for the created application.

3.1.3.2.2.5.1.5 Exceptions/Errors

The *create* operation shall raise the CreateApplicationRequestError exception when the input CF DeviceAssignmentSequence parameter contains one (1) or more invalid application component to device assignment(s).

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).

The *create* operation shall raise the InvalidInitConfiguration exception when the input initConfiguration parameter is invalid. The InvalidInitConfiguration invalidProperties parameter shall identify the invalid properties.

3.1.3.2.3 *DomainManager*

3.1.3.2.3.1 Description

The *DomainManager* interface is for the control and configuration of the system domain.

The *DomainManager* interface operations may be logically grouped into three categories: Human Computer Interface (HCI), Registration, and CF administration.

The HCI operations are used to configure the domain, get the domain's capabilities (devices, services, and applications), and initiate maintenance functions. Host operations are performed by an HCI client capable of interfacing to the domain manager.

The registration operations are used to register / unregister device managers, device manager's devices, device manager's services, and applications at startup or during run-time for dynamic device, service, and application extraction and insertion.

The administration operations are used to access the interfaces of registered device managers and domain manager's file manager.

3.1.3.2.3.2 UML

The DomainManager Interface UML is depicted in Figure 3-14.

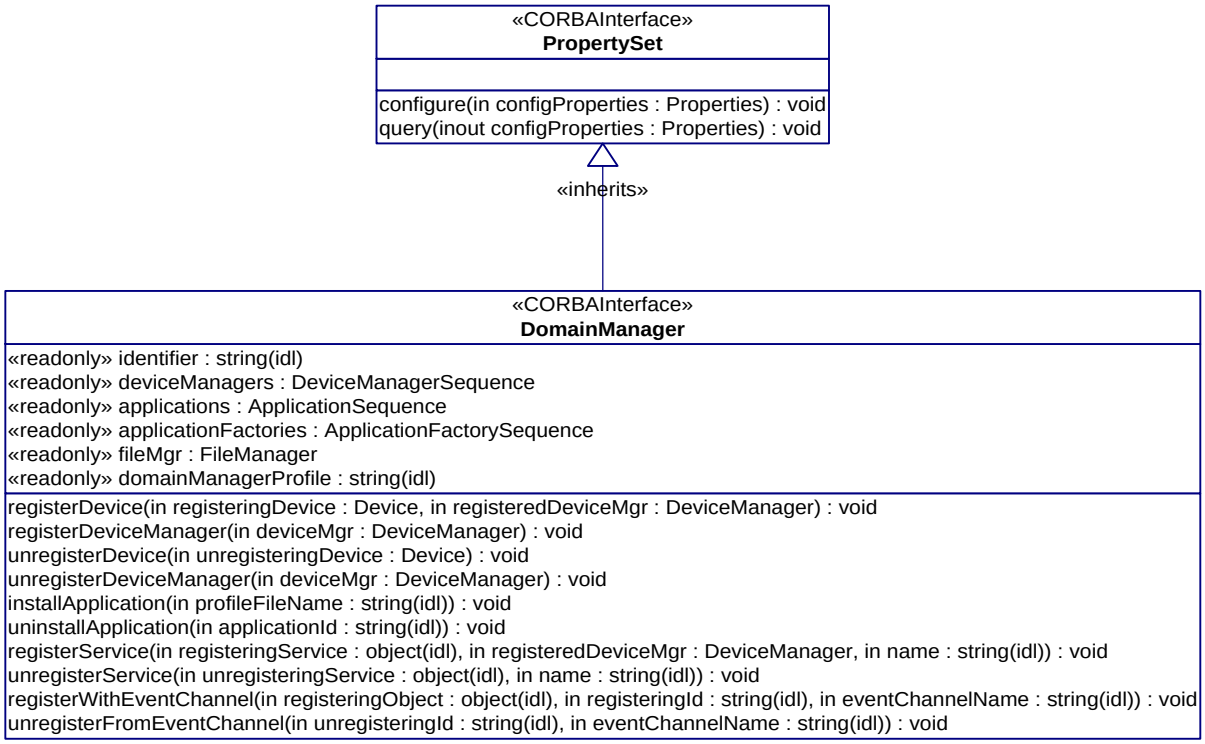

Figure 3-14: *DomainManager* Interface UML

3.1.3.2.3.3 Types

3.1.3.2.3.3.1 ApplicationInstallationError

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

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

3.1.3.2.3.3.2 InvalidIdentifier

The InvalidIdentifier exception indicates an application identifier is invalid.

```
exception InvalidIdentifier{};
```

3.1.3.2.3.3.3 DeviceManagerSequence

This type defines an unbounded sequence of *DeviceManager(s)*.

```
typedef sequence <DeviceManager> DeviceManagerSequence
```

3.1.3.2.3.3.4 ApplicationSequence

This type defines an unbounded sequence of *Application(s)*.

```
typedef sequence < Application> ApplicationSequence
```

3.1.3.2.3.3.5 ApplicationFactorySequence

This type defines an unbounded sequence of *ApplicationFactory(s)*.

```
typedef sequence < ApplicationFactory>
ApplicationFactorySequence
```

3.1.3.2.3.3.6 DeviceManagerNotRegistered Exception

The DeviceManagerNotRegistered exception indicates the registering device's device manager is not registered in the domain manager. A device's device manager has to be registered prior to device registration to the domain manager.

```
exception DeviceManagerNotRegistered{};
```

3.1.3.2.3.3.7 RegisterError

The RegisterError exception indicates that an internal error has occurred to prevent *DomainManager* registration operations from successful completion. The error number shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.2.3.3.8 UnregisterError

The UnregisterError exception indicates that an internal error has occurred to prevent *DomainManager* unregister operations from successful completion. The error number shall

indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.2.3.3.9 ApplicationUninstallationError

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

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

3.1.3.2.3.3.10 InvalidEventChannelName

The InvalidEventChannelName exception indicates that a domain manager was not able to locate the event channel.

```
exception InvalidEventChannelName{};
```

3.1.3.2.3.3.11 AlreadyConnected

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

```
exception AlreadyConnected{};
```

3.1.3.2.3.3.12 NotConnected

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

```
exception NotConnected{};
```

3.1.3.2.3.3.13 ApplicationAlreadyInstalled

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

```
exception ApplicationAlreadyInstalled{};
```

3.1.3.2.3.4 Attributes.

3.1.3.2.3.4.1 deviceManagers

The deviceManagers attribute is read-only containing a sequence of registered device managers in the domain. The readonly deviceManagers attribute shall contain a list of registered device managers that have registered with the domain manager. The domain manager shall write an ADMINISTRATIVE_EVENT log to a domain manager's log, when the deviceManagers attribute is obtained by a client.

```
readonly attribute DeviceManagerSequence deviceManagers;
```

3.1.3.2.3.4.2 applications

The applications attribute is read-only containing a sequence of instantiated *Applications* in the domain. The readonly applications attribute shall contain the list of *Applications* that have been instantiated. The domain manager shall write an ADMINISTRATIVE_EVENT log record to a domain manager's log, when the application's attribute is obtained by a client.

```
readonly attribute ApplicationSequence applications;
```

3.1.3.2.3.4.3 applicationFactories

The readonly applicationFactories attribute shall contain a list with one application factory per application (SAD file and associated files) successfully installed (i.e. no exception raised). The domain manager shall write an ADMINISTRATIVE_EVENT log record to a domain manager's log, when the applicationFactories attribute is obtained by a client.

```
readonly attribute ApplicationFactorySequence
applicationFactories;
```

3.1.3.2.3.4.4 fileMgr

The readonly fileMgr attribute shall contain the domain manager file manager. The domain manager shall write an ADMINISTRATIVE_EVENT log record to a domain manager's log, when the fileMgr attribute is obtained by a client.

```
readonly attribute FileManager fileMgr;
```

3.1.3.2.3.4.5 domainManagerProfile

The domainManagerProfile attribute contains the domain manager's Profile Descriptor.

The readonly domainManagerProfile attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the DomainManager Configuration Descriptor (DMD) file. Files referenced within the profile are obtained via the domain manager's *FileManager*.

```
readonly attribute string domainManagerProfile;
```

3.1.3.2.3.4.6 identifier

The readonly identifier attribute shall contain a unique identifier for a *DomainManager* instance. The identifier shall be identical to the *domainmanagerconfiguration* element *id* attribute of the domain manager's Descriptor (DMD) file.

```
readonly attribute string identifier;
```

3.1.3.2.3.5 General Class Behavior

The domain manager shall register itself with the CORBA Naming Service during component construction. The domain manager shall create a naming context using "/DomainName" as the *id* attribute to the input name parameter, and "" (Null string) as the *kind* attribute. The domain manager shall create a name binding to the created naming context using "/DomainName" as the *id* attribute to the input name parameter, and "" (Null string) as the *kind* attribute, where DomainName is identical to the *name* attribute of the domain manager's DMD domainmanagerconfiguration element and the input object parameter is the domain manager object reference. [6]

Since a log service is not a required component, a domain manager implementation may, or may not have access to a log. However, if log service(s) are available, a *DomainManager* implementation may use one or more of them. The logs utilized by the *DomainManager* implementation shall be defined in the DMD.

The domain manager shall begin to use a service specified in the DMD once the service is successfully registered with the domain manager via the *registerDeviceManager* or *registerService* operations.

The domain manager shall create its own *FileManager* component that consists of all registered device manager's *FileSystems.*

Upon system startup, the domain manager shall restore application factories for applications that were previously installed by the *DomainManager*::*installApplication* operation. The domain manager shall add the restored application factories to the *DomainManager* applicationFactories attribute.

The domain manager shall create the Incoming Domain Management and Outgoing Domain Management event channels.

3.1.3.2.3.6 Operations

3.1.3.2.3.6.1 *registerDeviceManager*

3.1.3.2.3.6.1.1 Brief Rationale

The *registerDeviceManager* operation is used to register a device manager, its device(s), and its services. Software profiles may be obtained from the device manager's *FileSystem.*

3.1.3.2.3.6.1.2 Synopsis

```
void registerDeviceManager (in DeviceManager deviceMgr) raises
(InvalidObjectReference, InvalidProfile, RegisterError );
```

3.1.3.2.3.6.1.3 Behavior

The *registerDeviceManager* operation verifies that the input deviceMgr parameter is a not a nil CORBA object reference.

The *registerDeviceManager* operation shall add the device manager indicated by the input deviceMgr parameter to the *DomainManager* deviceManagers attribute, if it does not already exist. The *registerDeviceManager* operation shall add the input device manager's registered devices and each registered device's attributes (e.g., identifier, softwareProfile, allocation properties, etc.) to the domain manager. The domain manager associates the input device manager's registered devices with the device manager in order to support the *unregisterDeviceManager* operation.

The *registerDeviceManager* operation shall add all the services contained in the registering device manager's *registeredServices* attribute to the domain manager. The *registerDeviceManager* operation associates the device manager indicated by the input deviceMgr parameter with its registered services in the domain manager in order to support the *unregisterDeviceManager* operation.

The *registerDeviceManager* operation shall return without exception and not register a new device manager when that device manager, indicated by the input deviceMgr parameter, has the

same identifier attribute as a previously registered device manager and the reference to the registered device manager refers to an existing object.

The *registerDeviceManager* operation shall register the new device manager indicated by the input deviceMgr parameter, when the previously registered device manager has the same identifier attribute as the new device manager and the reference to the registered device manager refers to a nonexistent object.

The *registerDeviceManager* operation shall write an ADMINISTRATIVE_EVENT log record when reference to the registered device manager refers to a nonexistent object.

The *registerDeviceManager* operation shall establish any connections for the device manager indicated by the input deviceMgr parameter, which are specified in the *connections* element of the device manager's Device Configuration Descriptor (DCD) file, that are possible with the current set of registered devices and services. Connections not currently possible are left unconnected pending future device / service registrations.

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

The *registerDeviceManager* operation shall obtain all the software profiles from the registering device manager's file systems.

The *registerDeviceManager* operation shall mount the device manager's file system to the domain manager's file manager. The mounted *FileSystem* name shall have the format, "/DomainName/HostName", where DomainName is the name of the domain and HostName is the input deviceMgr's label attribute.

The *registerDeviceManager* operation shall, upon unsuccessful device manager registration, write a FAILURE_ALARM log record to a domain manager's Log.

The *registerDeviceManager* operation shall send a DomainManagementObjectAddedEventType event to the Outgoing Domain Management event channel upon successful registration of a device manager. For this event,

1. The *producerId* is the identifier attribute of the domain manager.
2. The *sourceId* is the identifier attribute of the registered device manager.
3. The *sourceName* is the label attribute of the registered device manager.
4. The *sourceIOR* is the object reference for the registered device manager.
5. The *sourceCategory* is "DEVICE_MANAGER".

The following UML sequence diagram (Figure 3-15) illustrates the domain manager's behavior for the *registerDeviceManager* operation.

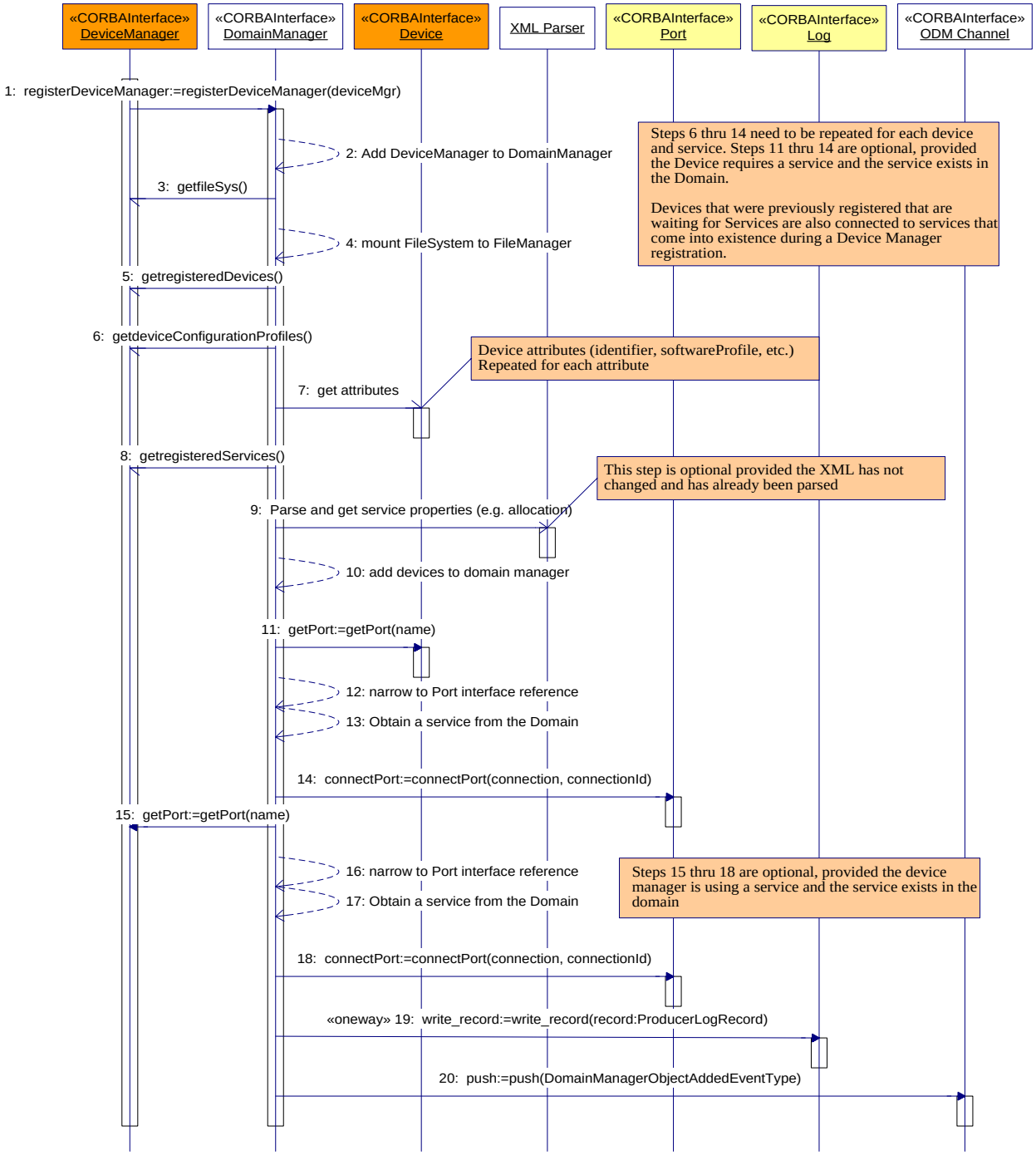

Figure 3-15: *DomainManager* Sequence Diagram for *registerDeviceManager* Operation

3.1.3.2.3.6.1.4 Returns

This operation does not return a value.

3.1.3.2.3.6.1.5 Exceptions/Errors

The *registerDeviceManager* operation shall raise the CF InvalidObjectReference exception when the input parameter *deviceMgr* contains an invalid reference to a *DeviceManager* interface.

The *registerDeviceManager* operation shall raise the CF InvalidProfile exception when the device manager's DCD file and the DCD's referenced files do not exist.

The *registerDeviceManager* operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

3.1.3.2.3.6.2 *registerDevice*

3.1.3.2.3.6.2.1 Brief Rationale

The *registerDevice* operation is used to register a device for a specific device manager with the domain manager.

3.1.3.2.3.6.2.2 Synopsis

```
void registerDevice (in Device registeringDevice, in
DeviceManager registeredDeviceMgr) raises
(InvalidObjectReference, InvalidProfile,
DeviceManagerNotRegistered, RegisterError);
```

3.1.3.2.3.6.2.3 Behavior

The *registerDevice* operation shall verify that the input parameters, registeringDevice and registeredDeviceMgr, are not nil CORBA object references.

The *registerDevice* operation shall add the device indicated by the input registeringDevice parameter and the device's attributes to the domain manager, if it does not already exist.

The *registerDevice* operation shall return without exception and not register a new device when that device, indicated by the input registeringDevice parameter, has the same identifier attribute as a previously registered device and the reference to the registered device refers to an existing object.

The *registerDevice* operation shall register the new device indicated by the input registeringDevice parameter, when the previously registered device has the same identifier attribute as the new device and the reference to the registered device refers to a nonexistent object.

The *registerDevice* operation shall write an ADMINISTRATIVE_EVENT log record when reference to the registered device refers to a nonexistent object.

The *registerDevice* operation associates the device indicated by the input registeringDevice parameter with the device manager indicated by the input registeredDeviceMgr parameter when the device manager is a valid registered *DeviceManager* in the domain manager.

The *registerDevice* operation shall establish any pending connections from previously registered device managers when the registering device completes these connections.

The *registerDevice* operation shall write an ADMINISTRATIVE_EVENT log record to a domain manager log upon successful device registration.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a domain manager log, when the CF InvalidProfile exception is raised.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a domain manager log when the DeviceManagerNotRegistered exception is raised.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a domain manager log when the CF InvalidObjectReference exception is raised.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a domain manager log when the RegisterError exception is raised.

The *registerDevice* operation shall send a DomainManagementObjectAddedEventType event to the Outgoing Domain Management event channel, upon successful registration of a device. For this event,

1. The *producerId* is the identifier attribute of the domain manager.
2. The *sourceId* is the identifier attribute of the registered device.
3. The *sourceName* is the label attribute of the registered device.
4. The *sourceIOR* is the object reference for the registered device.
5. The *sourceCategory* is “DEVICE”.

The following UML sequence diagram (Figure 3-16) illustrates the domain manager's behavior for the *registerDevice* operation.

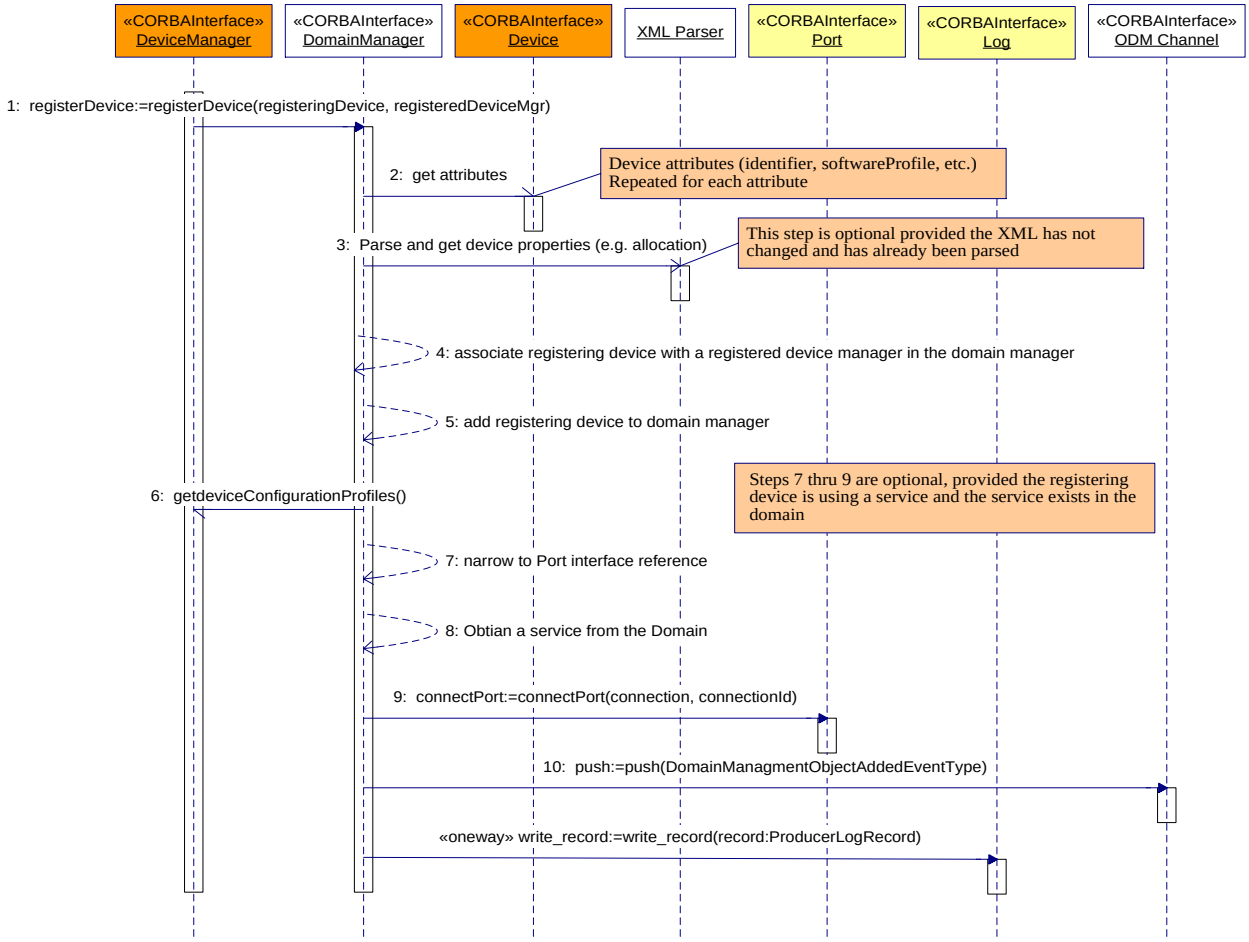

Figure 3-16: *DomainManager* Sequence Diagram for *registerDevice* Operation

3.1.3.2.3.6.2.4 Returns

This operation does not return a value.

3.1.3.2.3.6.2.5 Exceptions/Errors

The *registerDevice* operation shall raise the CF InvalidProfile exception when:

1. The device's SPD file and the SPD's referenced files do not exist, or
2. The device profile does not reference allocation properties.

The *registerDevice* operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr parameter is not a nil reference and the referenced device manager is not registered with the domain manager.

The *registerDevice* operation shall raise the CF InvalidObjectReference exception when input parameters registeringDevice or registeredDeviceMgr contains an invalid reference.

The *registerDevice* operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

3.1.3.2.3.6.3 *installApplication*

3.1.3.2.3.6.3.1 Brief Rationale

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

3.1.3.2.3.6.3.2 Synopsis

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

3.1.3.2.3.6.3.3 Behavior

The input profileFileName parameter is the absolute pathname of the application SAD.

The *installApplication* operation shall verify the existence of the application's SAD file and all files upon which the SAD depends, within the domain manager's file manager.

The *installApplication* operation shall write an ADMINISTRATIVE_EVENT log record to a domain manager's log, upon successful application installation.

The *installApplication* operation shall, upon unsuccessful application installation, write a FAILURE_ALARM log record to a domain manager's log.

The *installApplication* operation shall send a DomainManagementObjectAddedEventType 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 domain manager.
2. The *sourceId* is the identifier attribute of the installed application factory.
3. The *sourceName* is the name attribute of the installed application factory.
4. The *sourceIOR* is the object reference for the installed application factory.
5. The *sourceCategory* is “APPLICATION_FACTORY”.

3.1.3.2.3.6.3.4 Returns

This operation does not return a value.

3.1.3.2.3.6.3.5 Exceptions/Errors

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

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. The *installApplication* operation shall log a FAILURE_ALARM log record to a domain manager'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.

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

The *installApplication* operation shall write a FAILURE_ALARM log record to a domain manager'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.

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

3.1.3.2.3.6.4 *unregisterDeviceManager*

3.1.3.2.3.6.4.1 Brief Rationale

The *unregisterDeviceManager* operation is used to unregister a *DeviceManager* component from the domain manager. A device manager may be unregistered during run-time for dynamic extraction or maintenance of the device manager.

3.1.3.2.3.6.4.2 Synopsis

```
void unregisterDeviceManager (in DeviceManager deviceMgr) raises
(InvalidObjectReference, UnregisterError);
```

3.1.3.2.3.6.4.3 Behavior

The *unregisterDeviceManager* operation shall unregister a *DeviceManager* component from the *DomainManager*.

The *unregisterDeviceManager* operation shall release all device(s) and service(s) associated with the device manager that is being unregistered.

The *unregisterDeviceManager* operation shall disconnect the established connections (including those made to the CORBA Event Service event channels) of the unregistering device manager as well as for its registered devices and services. Connections broken as a result of the *unregisterDeviceManager* operation shall be considered as “pending” for future connections when the component to which the device manager or its registered devices and services were connected still exists. The *unregisterDeviceManager* operation may destroy the CORBA Event Service channel when no more consumers and producers are connected to it.

The *unregisterDeviceManager* operation shall unmount all device manager's file systems from its file manager.

The *unregisterDeviceManager* operation shall, upon the successful unregistration of a device manager, write an ADMINISTRATIVE_EVENT log record to a domain manager's log.

The *unregisterDeviceManager* operation shall, upon unsuccessful unregistration of a device manager, write a FAILURE_ALARM log record to a domain manager's log.

The *unregisterDeviceManager* operation shall send a DomainManagementObjectRemovedEventType event to the Outgoing Domain Management event channel, upon successful unregistration of a device manager. For this event,

1. The *producerId* is the identifier attribute of the domain manager.
2. The *sourceId* is the identifier attribute of the unregistered device manager.
3. The *sourceName* is the label attribute of the unregistered device manager.

4. The *sourceCategory* is “DEVICE_MANAGER”.

3.1.3.2.3.6.4.4 Returns

This operation does not return a value.

3.1.3.2.3.6.4.5 Exceptions/Errors

The *unregisterDeviceManager* operation shall raise the CF InvalidObjectReference when the input deviceMgr parameter contains an invalid reference to a *DeviceManager* interface.

The *unregisterDeviceManager* operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

3.1.3.2.3.6.5 *unregisterDevice*

3.1.3.2.3.6.5.1 Brief Rationale

The *unregisterDevice* operation is used to remove a device entry from the domain manager for a specific device manager.

3.1.3.2.3.6.5.2 Synopsis

```
void unregisterDevice (in Device unregisteringDevice) raises
(InvalidObjectReference, UnregisterError)
```

3.1.3.2.3.6.5.3 Behavior

The *unregisterDevice* operation shall remove a device entry from the domain manager.

The *unregisterDevice* operation shall release (client-side CORBA release) the unregisteringDevice from the domain manager.

The *unregisterDevice* operation shall disconnect the established connections (including those made to the CORBA Event Service event channels) of the unregistering device. Connections broken as a result of the *unregisterDevice* operation shall be considered as “pending” for future connections when the component to which the device was connected still exists.

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

The *unregisterDevice* operation shall, upon the successful unregistration of a device, write an ADMINISTRATIVE_EVENT log record to a domain manager's log.

The *unregisterDevice* operation shall, upon unsuccessful unregistration of a device, write a FAILURE_ALARM log record to a domain manager's log.

The *unregisterDevice* operation shall send a DomainManagementObjectRemovedEventType event to the Outgoing Domain Management event channel, upon successful unregistration of a device. For this event,

1. The *producerId* is the identifier attribute of the domain manager.
2. The *sourceId* is the identifier attribute of the unregistered device.
3. The *sourceName* is the label attribute of the unregistered device.
4. The *sourceCategory* is “DEVICE”.

3.1.3.2.3.6.5.4 Returns

This operation does not return a value.

3.1.3.2.3.6.5.5 Exceptions/Errors

The *unregisterDevice* operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a *Device* interface.

The *unregisterDevice* operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

3.1.3.2.3.6.6 *uninstallApplication*

3.1.3.2.3.6.6.1 Brief Rationale

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

3.1.3.2.3.6.6.2 Synopsis

```
void uninstallApplication (in string applicationId)raises
(InvalidIdentifier, ApplicationUninstallationError);
```

3.1.3.2.3.6.6.3 Behavior

The ApplicationId parameter is the *softwareassembly* element *id* attribute of the *ApplicationFactory's* Software Assembly Descriptor file.

The *uninstallApplication* operation shall make the *ApplicationFactory* unavailable from the domain manager (i.e. its services no longer provided for the application).

The *uninstallApplication* operation shall, upon successful uninstall of an application, write an ADMINISTRATIVE_EVENT log record to a domain manager's log.

The *uninstallApplication* operation shall, upon unsuccessful uninstall of an application, write a FAILURE_ALARM log record to a domain manager's log.

The *uninstallApplication* operation shall send a DomainManagementObjectRemovedEventType 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 domain manager.
2. The *sourceId* is the identifier attribute of the uninstalled application factory.
3. The *sourceName* is the name attribute of the uninstalled application factory.
4. The *sourceCategory* is "APPLICATION_FACTORY".

3.1.3.2.3.6.6.4 Returns

This operation does not return a value.

3.1.3.2.3.6.6.5 Exceptions/Errors

The *uninstallApplication* operation shall raise the InvalidIdentifier exception when the ApplicationId is invalid.

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

3.1.3.2.3.6.7 *registerService*

3.1.3.2.3.6.7.1 Brief Rationale

The *registerService* operation is used to register a service for a specific device manager with the domain manager.

3.1.3.2.3.6.7.2 Synopsis

```
void registerService (in Object registeringService, in
DeviceManager registeredDeviceMgr, in string name) raises
(InvalidObjectReference, DeviceManagerNotRegistered,
RegisterError);
```

3.1.3.2.3.6.7.3 Behavior

The *registerService* operation shall verify the input registeringService and registeredDeviceMgr are valid object references.

The *registerService* operation shall verify the input registeredDeviceMgr has been previously registered with the domain manager.

The *registerService* operation shall add the registeringService's object reference and the *registeringService's* name to the domain manager, if the name for the type of service being registered does not exist within the domain manager. The *registerService* operation shall return without exception and not register a new service when that service, indicated by the input registeringService parameter, has the same name and type as a previously registered service and the reference to the registered service refers to an existing object.

The *registerService* operation shall register the new service, indicated by the input registeringService parameter, when the previously registered service has the same name and type as the new service and the reference to the registered service refers to a nonexistent object.

The *registerService* operation shall write an ADMINISTRATIVE_EVENT log record when reference to the registered service refers to a nonexistent object.

The *registerService* operation shall associate the input *registeringService* parameter with the input *registeredDeviceMgr* parameter in the domain manager, when the *registeredDeviceMgr* parameter indicates a device manager that is registered with the domain manager.

The *registerService* operation shall establish any pending connections from previously registered device managers when the registering service completes these connections.

The *registerService* operation shall, upon successful service registration, write an ADMINISTRATIVE_EVENT log record to a domain manager's log.

The *registerService* operation shall, upon unsuccessful service registration, write a FAILURE_ALARM log record to a domain manager's log.

The *registerService* operation shall send a DomainManagementObjectAddedEventType event to the Outgoing Domain Management event channel, upon successful registration of a service. For this event,

1. The *producerId* is the identifier attribute of the domain manager.

2. The *sourceId* is the identifier attribute of the *componentinstantiation* element associated with the registered service.
3. The *sourceName* is the input name parameter for the registering service.
4. The *sourceIOR* is the object reference for the registered service.
5. The *sourceCategory* is "SERVICE".

The following UML sequence diagram (Figure 3-17) illustrates the domain manager's behavior for the *registerService* operation.

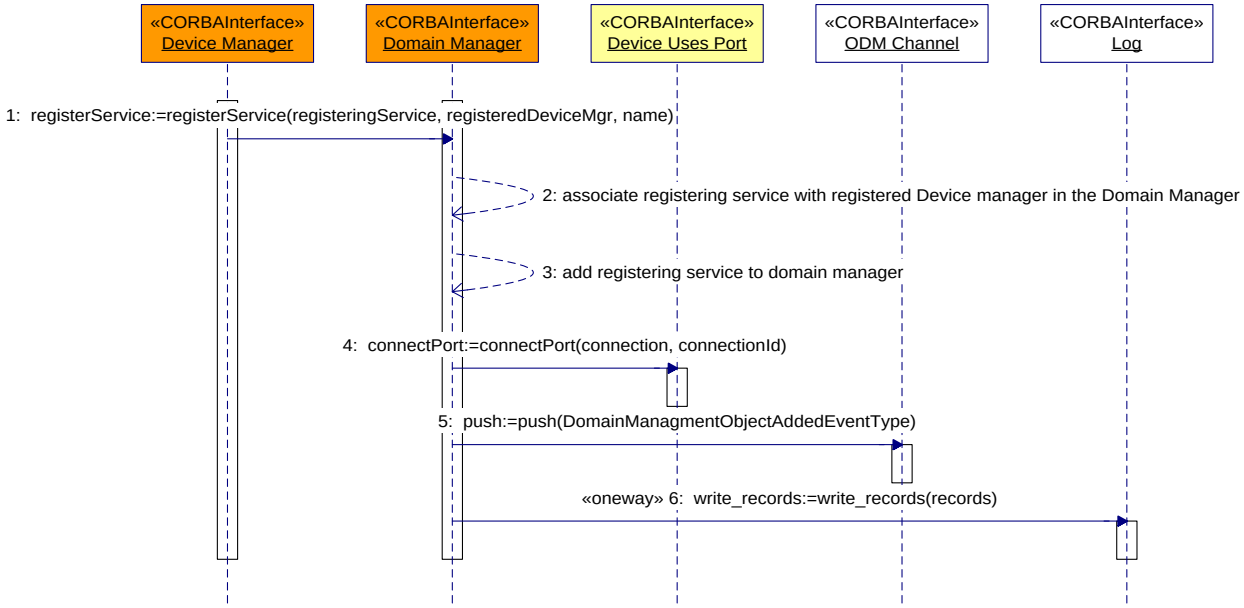

Figure 3-17: DomainManager Sequence Diagram for *registerService* Operation

3.1.3.2.3.6.7.4 Returns

This operation does not return a value.

3.1.3.2.3.6.7.5 Exceptions/Errors.

The *registerService* operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr parameter is not a nil reference and is not registered with the domain manager.

The *registerService* operation shall raise the CF InvalidObjectReference exception when input parameters registeringService or registeredDeviceMgr contains an invalid reference.

The *registerService* operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

3.1.3.2.3.6.8 *unregisterService*

3.1.3.2.3.6.8.1 Brief Rationale

The *unregisterService* operation is used to remove a service entry from the domain manager for a specific device manager.

3.1.3.2.3.6.8.2 Synopsis

```
void unregisterService (in Object unregisteringService, in
string name) raises (InvalidObjectReference, UnregisterError);
```

3.1.3.2.3.6.8.3 Behavior

The *unregisterService* operation shall disconnect the established connections (including those made to the CORBA Event Service event channels) of the unregistering service indicated by the input unregisteringService parameter. Connections broken as a result of the *unregisterService* operation shall be considered as "pending" for future connections when the component to which the service was connected still exists.

The *unregisterService* operation shall remove the unregisteringService entry specified by the input name parameter from the domain manager.

The *unregisterService* operation shall release (client-side CORBA release) the unregisteringService from the domain manager.

The *unregisterService* operation shall, upon the successful unregistration of a service, write an ADMINISTRATIVE_EVENT log record to a domain manager's log.

The *unregisterService* operation shall, upon unsuccessful unregistration of a service, write a FAILURE_ALARM log record to a domain manager's log.

The *unregisterService* operation shall send a DomainManagementObjectRemovedEventType event to the Outgoing Domain Management event channel, upon successful unregistration of a service. For this event,

1. The *producerId* is the identifier attribute of the domain manager.
2. The *sourceId* is the identifier attribute of the *componentinstantiation* element associated with the unregistered service.
3. The *sourceName* is the input name parameter for the unregistering service.
4. The *sourceCategory* is "SERVICE".

3.1.3.2.3.6.8.4 Returns

This operation does not return a value.

3.1.3.2.3.6.8.5 Exceptions/Errors

The *unregisterService* operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a service interface.

The *unregisterService* operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

3.1.3.2.3.6.9 *registerWithEventChannel*

3.1.3.2.3.6.9.1 Brief Rationale

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

3.1.3.2.3.6.9.2 Synopsis

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

3.1.3.2.3.6.9.3 Behavior

The *registerWithEventChannel* operation shall connect the object identified by the input registeringObject parameter to an event channel as specified by the input eventChannelName parameter.

3.1.3.2.3.6.9.4 Returns

This operation does not return a value.

3.1.3.2.3.6.9.5 Exceptions/Errors

The *registerWithEventChannel* operation shall raise the CF InvalidObjectReference exception when the input registeringObject parameter contains an invalid reference to a *CosEventComm::PushConsumer* interface.

The *registerWithEventChannel* operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter contains an invalid event channel name.

The *registerWithEventChannel* operation shall raise AlreadyConnected exception when the input parameter contains a connection to the event channel for the input registeringId parameter.

3.1.3.2.3.6.10 *unregisterFromEventChannel*

3.1.3.2.3.6.10.1 Brief Rationale

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

3.1.3.2.3.6.10.2 Synopsis

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

3.1.3.2.3.6.10.3 Behavior

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

3.1.3.2.3.6.10.4 Returns

This operation does not return a value.

3.1.3.2.3.6.10.5 Exceptions/Errors

The *unregisterFromEventChannel* operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter contains an invalid reference to an event channel.

The *unregisterFromEventChannel* operation shall raise the NotConnected exception when the input parameter unregisteringId parameter is not connected to specified input event channel.

3.1.3.2.4 *DeviceManager*

3.1.3.2.4.1 Description

The *DeviceManager* interface is used to manage a set of logical devices and services. The interface for a *DeviceManager* is based upon its attributes, which are:

1. Device Configuration Profile - a mapping of physical device locations to meaningful labels (e.g., audio1, serial1, etc.), along with the devices and services to be deployed.
2. File System - the FileSystem associated with this device manager.
3. Device Manager Identifier - the instance-unique identifier for this device manager.
4. Device Manager Label - the meaningful name given to this device manager.
5. Registered Devices - a list of devices that have registered with this device manager.
6. Registered Services - a list of services that have registered with this device manager.

3.1.3.2.4.2 UML

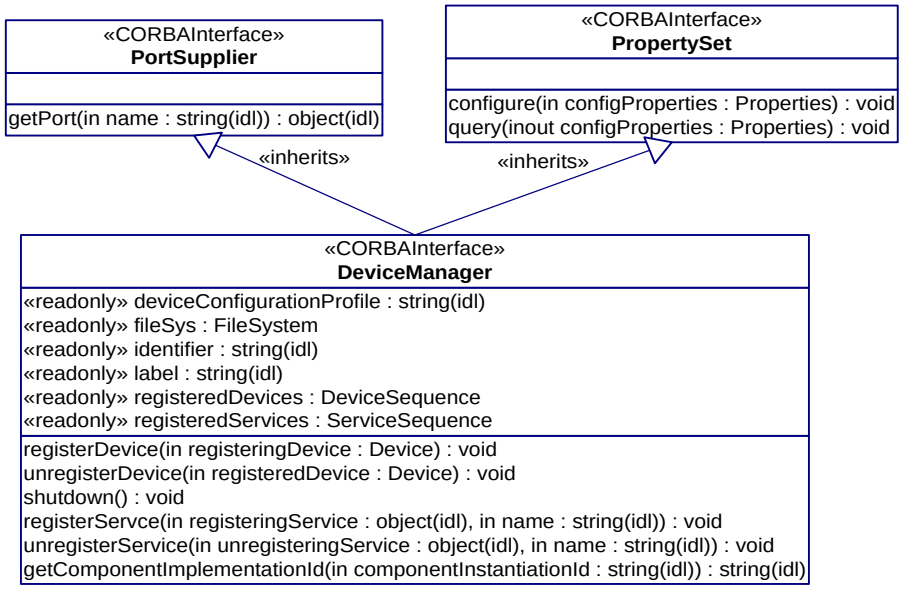

Figure 3-18: *DeviceManager* UML

3.1.3.2.4.3 Types

This section describes the types defined in the interface *DeviceManager*.

3.1.3.2.4.3.1 ServiceType

This structure provides the object reference and name of a service that has registered with the device manager.

```
struct ServiceType
{
     Object serviceObject;
     string serviceName;
};
```

3.1.3.2.4.3.2 ServiceSequenceType

This type provides an unbounded sequence of ServiceType structures for services that have registered with the device manager.

```
typedef sequence <ServiceType> ServiceSequence;
```

3.1.3.2.4.4 Attributes

3.1.3.2.4.4.1 identifier

The readonly identifier attribute shall contain the instance-unique identifier for a device manager. The identifier shall be identical to the *deviceconfiguration* element *id* attribute of the device manager's Device Configuration Descriptor (DCD) file.

```
readonly attribute string identifier;
```

3.1.3.2.4.4.2 label

The readonly label attribute shall contain the device manager's label. The label is the meaningful name given to a device manager.

```
readonly attribute string label;
```

3.1.3.2.4.4.3 fileSys

The readonly fileSys attribute shall contain the *FileSystem* associated with this device manager.

```
readonly attribute FileSystem fileSys;
```

3.1.3.2.4.4.4 deviceConfigurationProfile

The readonly deviceConfigurationProfile attribute contains the device manager's profile descriptor.

The readonly deviceConfigurationProfile attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the device manager's Device Configuration Descriptor (DCD) file. Files referenced within the profile are obtained via the *FileSystem*.

```
readonly attribute string deviceConfigurationProfile;
```

3.1.3.2.4.4.5 registeredDevices

The readonly registeredDevices attribute shall contain a list of devices that have registered with this device manager or a sequence length of zero if no devices have registered with the device manager.

```
readonly attribute DeviceSequence registeredDevices;
```

3.1.3.2.4.4.6 registeredServices

The readonly registeredServices attribute shall contain a list of services that have registered with this device manager or a sequence length of zero if no services have registered with the device manager.

```
readonly attribute ServiceSequence registeredServices;
```

3.1.3.2.4.5 General Behavior

The device manager upon start up shall register itself with a domain manager. This requirement allows the system to be developed where at a minimum only the *DomainManager's* object reference needs to be known. A device manager shall use the information in the device manager's DCD for determining:

1. Services to be deployed for this device manager (for example, *log*(s)),
2. Devices to be created for this device manager (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. Devices to be deployed on (executing on) another device,
4. Devices to be aggregated to another device,
5. Mount point names for file systems,
6. The *DeviceManager's* identifier attribute value which is the DCD's *id* attribute value, and
7. The *DeviceManager's* label attribute value which is the DCD's name attribute value.

The device manager shall create *FileSystem* components implementing the *FileSystem* interface for each OS file system. If multiple file systems are to be created, the device manager shall mount created file systems to a *FileManager* component (widened to a *FileSystem* through the FileSys attribute). The mount points used for the created file systems are identical to the values identified in the *filesystemnames* element of the device manager's Device Configuration Descriptor. Each mounted file system name shall be unique within the device manager.

The device manager shall supply execute operation parameters for a device consisting of:

1. Device manager IOR – The ID is "DEVICE_MGR_IOR" and the value is a string that is the *DeviceManager* stringified IOR.
2. Profile Name – The ID is "PROFILE_NAME" and the value is a CORBA string that is the full mounted file system file path name.
3. Device Identifier – The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD *componentinstantiation id* attribute.
4. Device Label – The ID is "DEVICE_LABEL" and the value is a string that corresponds to the DCD *componentinstantiation usage* element. This parameter is only used when the DCD *componentinstantiation usage* element is specified.
5. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an *AggregateDevice* stringified IOR. This parameter is only used

when the DCD *componentinstantiation* element represents the child device of another *componentinstantiation* element.

6. The execute ("execparam") properties as specified in the DCD for a *componentinstantiation* element. The device manager shall pass the componentinstantiation element "execparam" properties that have values as parameters. The device manager shall pass "execparam" parameters' IDs and values as string values.

The device manager shall use the *componentinstantiation* element's SPD *implementation* code's *stacksize* and *priority* elements, when specified, for the *execute* operation options parameters.

The device manager shall initialize and then configure logical devices that are started by the device manager, after they have successfully registered with the device manager. The device manager shall configure a DCD's *componentinstantiation* element provided the *componentinstantiation* element has "configure" readwrite or writeonly properties with values. Figure 3-19 depicts a device manager startup scenario.

If a service is deployed by the device manager, the device manager shall supply execute operation parameters consisting of:

1. Device manager IOR – The ID is "DEVICE_MGR_IOR" and the value is a string that is the *DeviceManager* stringified IOR.
2. Service Name – The ID is "SERVICE_NAME" and the value is a string that corresponds to the DCD *componentinstantiation usagename* element.
3. The execute ("execparam") properties as specified in the DCD for a *componentinstantiation* element. The device manager shall pass the componentinstantiation element "execparam" properties that have values as parameters. The device manager shall pass "execparam" parameters' IDs and values as string values.

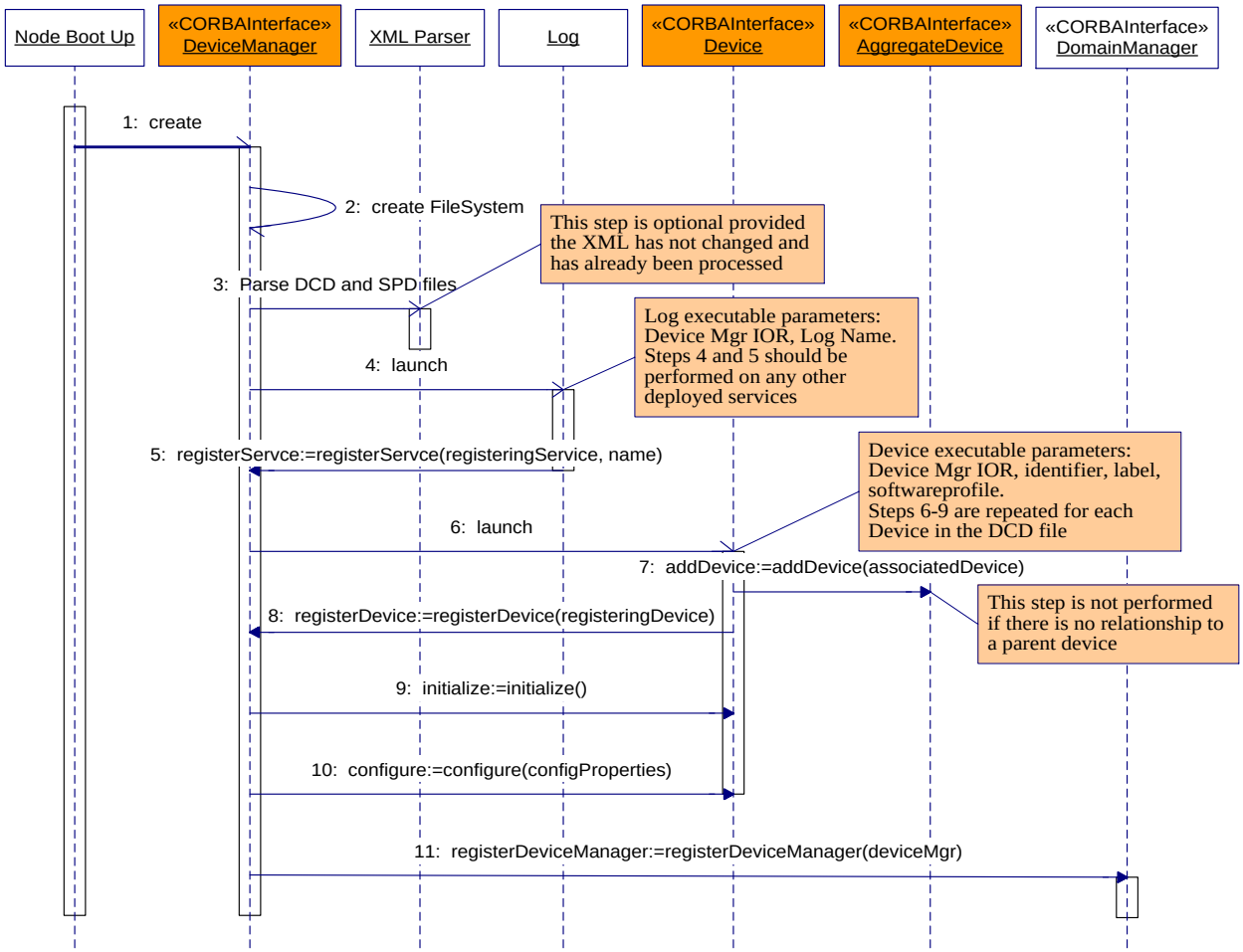

Figure 3-19: Device Manager Startup Scenario

3.1.3.2.4.6 Operations

3.1.3.2.4.6.1 registerDevice

3.1.3.2.4.6.1.1 Brief Rationale

The *registerDevice* operation provides the mechanism to register a device with a device manager.

3.1.3.2.4.6.1.2 Synopsis

```
void registerDevice (in Device registeringDevice) raises
(InvalidObjectReference);
```

3.1.3.2.4.6.1.3 Behavior

The *registerDevice* operation shall add the input registeringDevice to the *DeviceManager* registeredDevices attribute when the input registeringDevice does not already exist in the registeredDevices attribute. The registeringDevice is ignored when duplicated.

The *registerDevice* operation shall register the registeringDevice with the domain manager when the device manager has already registered to the domain manager and the registeringDevice has been successfully added to the *DeviceManager* registeredDevices attribute.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a domain manager's log, upon unsuccessful registration of a device to the *DeviceManager* registeredDevices attribute.

3.1.3.2.4.6.1.4 Returns

This operation does not return any value.

3.1.3.2.4.6.1.5 Exceptions/Errors

The *registerDevice* operation shall raise the CF InvalidObjectReference when the input registeringDevice is a nil CORBA object reference.

3.1.3.2.4.6.2 unregisterDevice

3.1.3.2.4.6.2.1 Brief Rationale

The *unregisterDevice* operation unregisters a device from a device manager.

3.1.3.2.4.6.2.2 Synopsis

```
void unregisterDevice (in Device registeredDevice) raises
(InvalidObjectReference);
```

3.1.3.2.4.6.2.3 Behavior

The *unregisterDevice* operation shall remove the input registeredDevice from the *DeviceManager* registeredDevices attribute. The *unregisterDevice* operation shall unregister the input registeredDevice from the domain manager when the input registeredDevice is registered with the device manager and the device manager is not shutting down.

The *unregisterDevice* operation shall write a FAILURE_ALARM log record, when it cannot successfully remove a registeredDevice from the *DeviceManager* registeredDevices attribute.

3.1.3.2.4.6.2.4 Returns

This operation does not return any value.

3.1.3.2.4.6.2.5 Exceptions/Errors

The *unregisterDevice* operation shall raise the CF InvalidObjectReference when the input registeredDevice is a nil CORBA object reference or does not exist in the *DeviceManager's* registeredDevices attribute.

3.1.3.2.4.6.3 registerService

3.1.3.2.4.6.3.1 Brief Rationale

The *registerService* operation provides the mechanism to register a service with a device manager.

3.1.3.2.4.6.3.2 Synopsis

```
void registerService (in Object registeringService, in string
name) raises (InvalidObjectReference);
```

3.1.3.2.4.6.3.3 Behavior

The *registerService* operation shall add the input registeringService to the *DeviceManager* registeredServices attribute when the input registeringService does not already exist in the registeredServices attribute. The registeringService is ignored when duplicated.

The *registerService* operation shall register the registeringService with the domain manager when the device manager has already registered to the domain manager and the registeringService has been successfully added to the *DeviceManager's* registeredServices attribute.

The *registerService* operation shall write a FAILURE_ALARM log record, upon unsuccessful registration of a service to the *DeviceManager* registeredServices attribute.

3.1.3.2.4.6.3.4 Returns

This operation does not return any value.

3.1.3.2.4.6.3.5 Exceptions/Errors

The *registerService* operation shall raise the CF InvalidObjectReference exception when the input registeringService is a nil CORBA object reference.

3.1.3.2.4.6.4 unregisterService

3.1.3.2.4.6.4.1 Brief Rationale.

The *unregisterService* operation unregisters a service from a device manager.

3.1.3.2.4.6.4.2 Synopsis

```
void unregisterService (in Object unregisteringService, in
string name) raises (InvalidObjectReference);
```

3.1.3.2.4.6.4.3 Behavior

The *unregisterService* operation shall remove the input registered service specified by the input *name* parameter from the *DeviceManager::registeredServices* attribute. The *unregisterService* operation shall unregister the input unregistering service from the domain manager when the device manager is not in the SHUTTING_DOWN state.

The *unregisterService* operation shall write a FAILURE_ALARM log record, when it cannot successfully remove a registeredService from the *DeviceManager registeredServices* attribute.

3.1.3.2.4.6.4.4 Returns

This operation does not return any value.

3.1.3.2.4.6.4.5 Exceptions/Errors

The *unregisterService* operation shall raise the CF InvalidObjectReference when the input unregistering service is a nil CORBA object reference or does not exist in the *DeviceManager registeredServices* attribute.

3.1.3.2.4.6.5 *shutdown*

3.1.3.2.4.6.5.1 Brief Rationale

The *shutdown* operation provides the mechanism to terminate a device manager.

3.1.3.2.4.6.5.2 Synopsis

```
void shutdown();
```

3.1.3.2.4.6.5.3 Behavior

The *shutdown* operation shall unregister the device manager from the domain manager.

The *shutdown* operation shall perform releaseObject on all of the device manager's registered devices (*DeviceManager* registeredDevices attribute).

The *shutdown* operation shall cause the device manager to be unavailable (i.e. released from the CORBA environment and its process terminated on the OS), when all of the device manager's registered devices are unregistered from the device manager.

3.1.3.2.4.6.5.4 Returns

This operation does not return any value.

3.1.3.2.4.6.5.5 Exceptions/Errors

This operation does not raise any exceptions.

3.1.3.2.4.6.6 *getComponentImplementationId*.

3.1.3.2.4.6.6.1 Brief Rational

The *getComponentImplementationId* operation returns the SPD implementation ID that the *DeviceManager* interface used to create a component.

3.1.3.2.4.6.6.2 Synopsis

```
string getComponentImplementationId (in string
componentInstantiationId);
```

3.1.3.2.4.6.6.3 Behavior

The *getComponentImplementationId* operation returns the SPD *implementation* element's *id* attribute that matches the *id* attribute of the SPD *implementation* element used to create the component specified by the input componentInstantiationId parameter.

3.1.3.2.4.6.6.4 Returns

The *getComponentImplementationId* operation shall return the SPD *implementation* element's *id* attribute that matches the SPD *implementation* element used to create the component identified by the input componentInstantiationId parameter. The *getComponentImplementationId* operation shall return an empty string when the input componentInstantiationId parameter does not match the *id* attribute of any SPD *implementation* element used to create the component.

3.1.3.2.4.6.6.5 Exceptions/Errors

This operation does not raise any exceptions.

3.1.3.3 Base Device 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 *Device*, *LoadableDevice* and *ExecutableDevice*.

Base Device Interfaces shall be implemented using the CF IDL presented in Appendix C.

3.1.3.3.1 *Device*

3.1.3.3.1.1 Description

A device is a type of resource and has all the requirements associated with the *Resource* interface. The *Device* interface defines additional capabilities and attributes for any logical device in the domain. A logical device is a software abstraction for a physical hardware device and provides the following attributes and operations:

1. Software Profile Attribute – The SPD referenced by this *profile* element (Profile Descriptor) defines the logical device capabilities (data/command uses and provides ports, configure and query properties, capacity properties, status properties, etc.), which could be a subset of the hardware device's capabilities.
2. State Management & Status Attributes – This information describes the administrative, usage, and operational states of the device.
3. 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 software profile.

3.1.3.3.1.2 UML

The *Device* Interface UML is depicted in Figure 3-20.

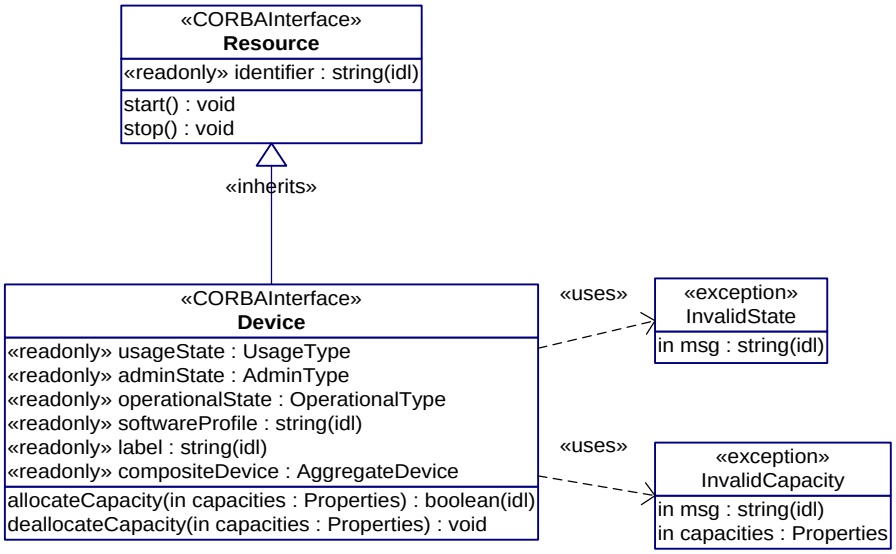

Figure 3-20: *Device* Interface UML

3.1.3.3.1.3 Types

3.1.3.3.1.3.1 InvalidState

The InvalidState exception indicates that the device is not capable of the behavior being attempted due to the state the device is in.

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

3.1.3.3.1.3.2 InvalidCapacity

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

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

3.1.3.3.1.3.3 AdminType

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

```
enum AdminType
{
      LOCKED,
      SHUTTING_DOWN,
      UNLOCKED
};
```

3.1.3.3.1.3.4 OperationalType

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

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

3.1.3.3.1.3.5 UsageType

This is a CORBA 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

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

3.1.3.3.1.4 Attributes

3.1.3.3.1.4.1 usageState.

The readonly usageState attribute shall contain 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.

The device shall send a StateChangeEventType 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 device.
2. The *sourceId* field is the identifier attribute of the device.
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.

```
readonly attribute UsageType usageState;
```

3.1.3.3.1.4.2 adminState

The administrative state indicates the permission to use or prohibition against using the device. The adminState attribute shall contain the device's admin state value. 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 transitions commands are ignored.

The adminState attribute, upon being commanded to be LOCKED, shall transition from the UNLOCKED to the SHUTTING_DOWN state and set the adminState to LOCKED for its entire aggregation of devices (if it has any). The adminState shall then transition to the LOCKED state when the device's usageState is IDLE and its entire aggregation of devices are LOCKED. Refer to Figure 3-21 for an illustration of the above state behavior.

The device shall send a StateChangeEventType 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 device.
2. The *sourceId* field is the identifier attribute of the device.
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.

```
attribute AdminType adminState;
```

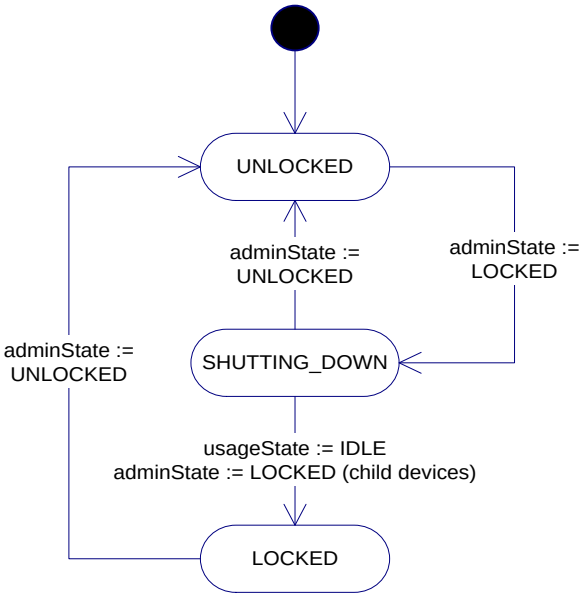

Figure 3-21: State Transition Diagram for adminState

3.1.3.3.1.4.3 operationalState

The readonly operationalState attribute shall contain the device's operational state (ENABLED or DISABLED). The operational state indicates whether or not the device is functioning.

The device 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 device.
2. The *sourceId* field is the identifier attribute of the device.
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.

```
readonly attribute OperationalType operationalState;
```

3.1.3.3.1.4.4 softwareProfile

The softwareProfile attribute contains the profile descriptor for this device.

The readonly softwareProfile attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the SPD file. Files referenced within the profile are obtained via the *FileManager*.

```
readonly attribute string softwareProfile;
```

3.1.3.3.1.4.5 label

The readonly label attribute shall contain the device's label. The label attribute is the meaningful name given to a device. The attribute could convey location information within the system (e.g., audio1, serial1, etc.).

```
readonly attribute string label;
```

3.1.3.3.1.4.6 compositeDevice

The readonly compositeDevice attribute shall contain the object reference of the aggregate device when this device is a parent device. The readonly compositeDevice attribute shall contain a nil CORBA object reference when this device is not a parent device.

```
readonly attribute AggregateDevice compositeDevice;
```

3.1.3.3.1.5 Operations

3.1.3.3.1.5.1 *allocateCapacity*

3.1.3.3.1.5.1.1 Brief Rationale

The *allocateCapacity* operation provides the mechanism to request and allocate capacity from the *Device*.

3.1.3.3.1.5.1.2 Synopsis

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

3.1.3.3.1.5.1.3 Behavior

The *allocateCapacity* operation shall reduce the current capacities of the device based upon the input capacities parameter, when the device's adminState is UNLOCKED, device's operationalState is ENABLED, and device's usageState is not BUSY.

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. The *allocateCapacity* operation shall set the usageState attribute to ACTIVE, when capacity is being used and any capacity is still available for allocation (reference Figure 3-22).

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.

3.1.3.3.1.5.1.4 Returns

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

3.1.3.3.1.5.1.5 Exceptions/Errors

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.

The *allocateCapacity* operation shall raise the InvalidState exception, when the *Device's* adminState is not UNLOCKED or operationalState is DISABLED.

3.1.3.3.1.5.2 *deallocateCapacity*

3.1.3.3.1.5.2.1 Brief Rationale

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

3.1.3.3.1.5.2.2 Synopsis

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

3.1.3.3.1.5.2.3 Behavior

The *deallocateCapacity* operation shall adjust the current capacities of the device based upon the input capacities parameter.

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

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

The *deallocateCapacity* operation shall set the adminState attribute to LOCKED as specified in 3.1.3.2.4.4.2.

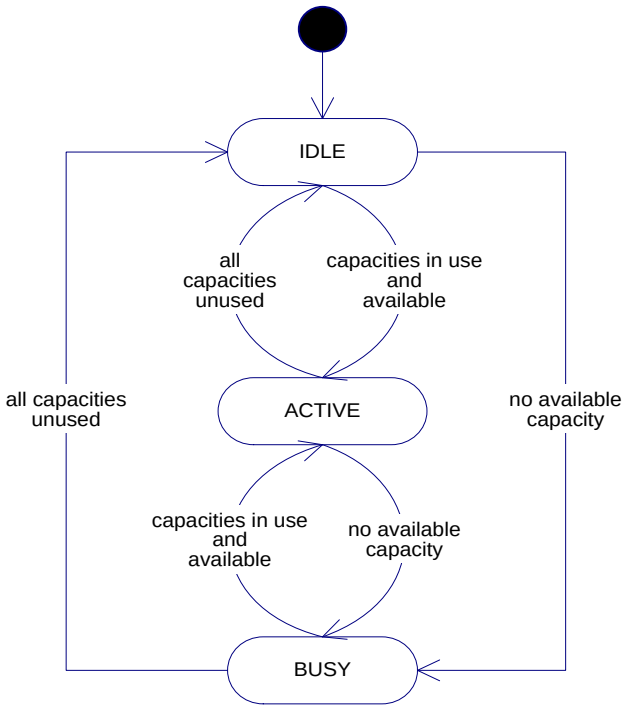

Figure 3-22: State Transition Diagram for *allocateCapacity* and *deallocateCapacity*

3.1.3.3.1.5.2.4 Returns

This operation does not return any value.

3.1.3.3.1.5.2.5 Exceptions/Errors

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.

The *deallocateCapacity* operation shall raise the InvalidState exception, when the device's adminState is LOCKED or operationalState is DISABLED.

3.1.3.3.1.5.3 *releaseObject*

3.1.3.3.1.5.3.1 Description

This section describes additional release behavior for a logical device.

3.1.3.3.1.5.3.2 Synopsis

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

3.1.3.3.1.5.3.3 Behavior

The following behavior is in addition to the *LifeCycle::releaseObject* operation behavior.

The *releaseObject* operation shall assign the LOCKED state to the *Device* adminState attribute, when the *Device* adminState attribute is UNLOCKED.

The *releaseObject* operation shall call the *releaseObject* operation on all those devices that are contained within the *AggregateDevice* devices attribute, when this device is a parent device.

The *releaseObject* operation shall cause the removal of the device from the *Device* compositeDevice attribute, when this device is a child device.

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

The *releaseObject* operation shall unregister its device from its device manager.

The following three figures (Figure 3-23, Figure 3-24, and Figure 3-25) depict different release scenarios depending on the type of device and the state the device is in.

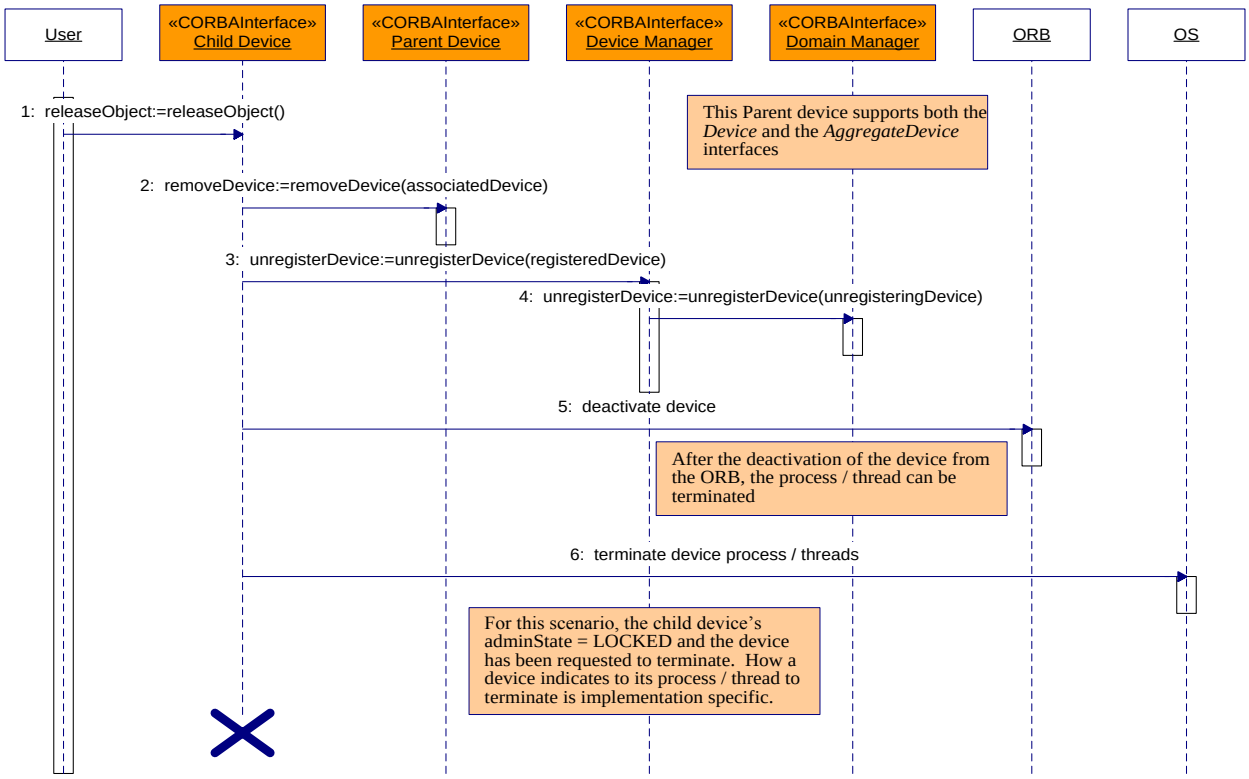

Figure 3-23: Release Aggregated *Device* Scenario

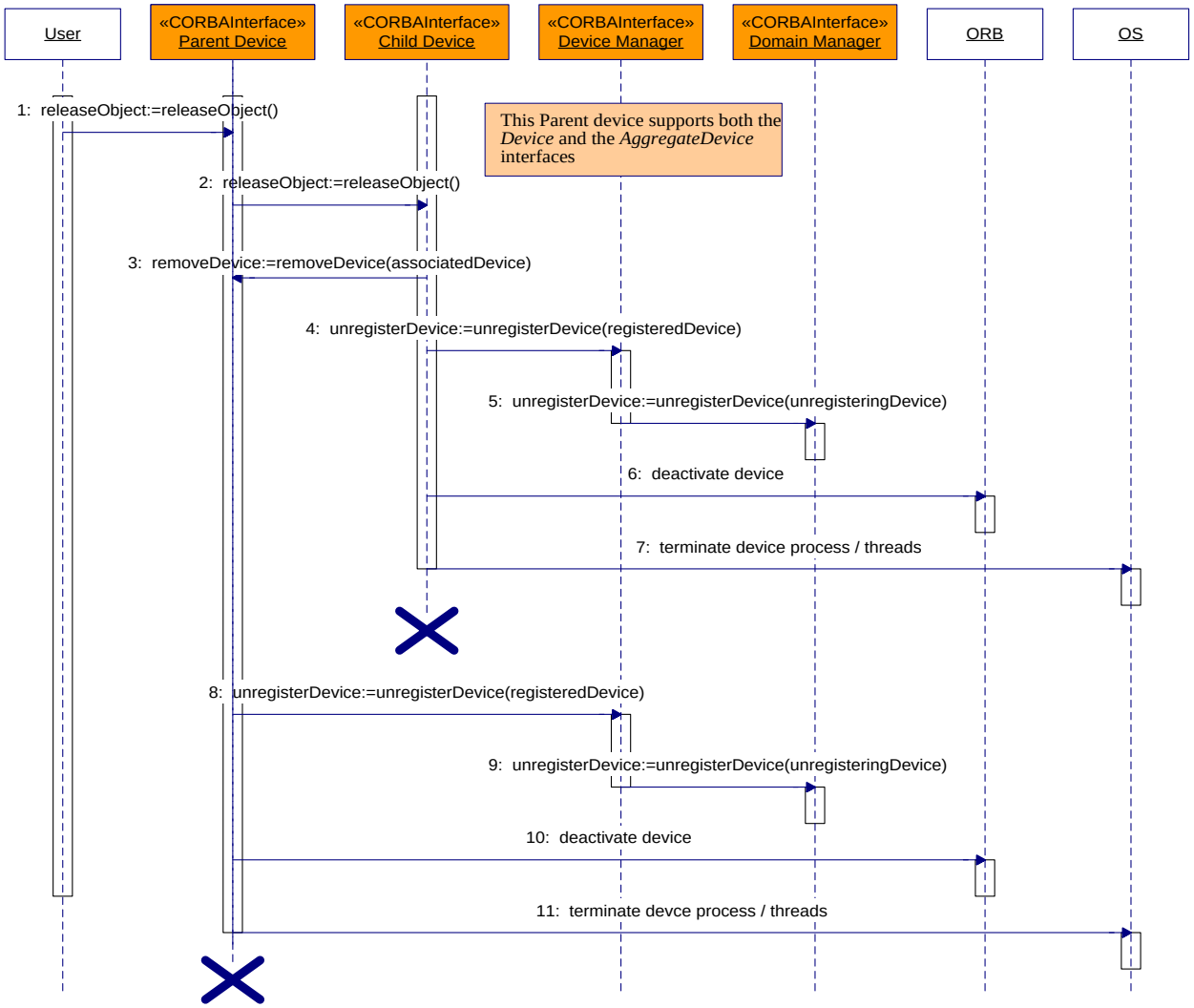

Figure 3-24: Release Composite *Device* Scenario

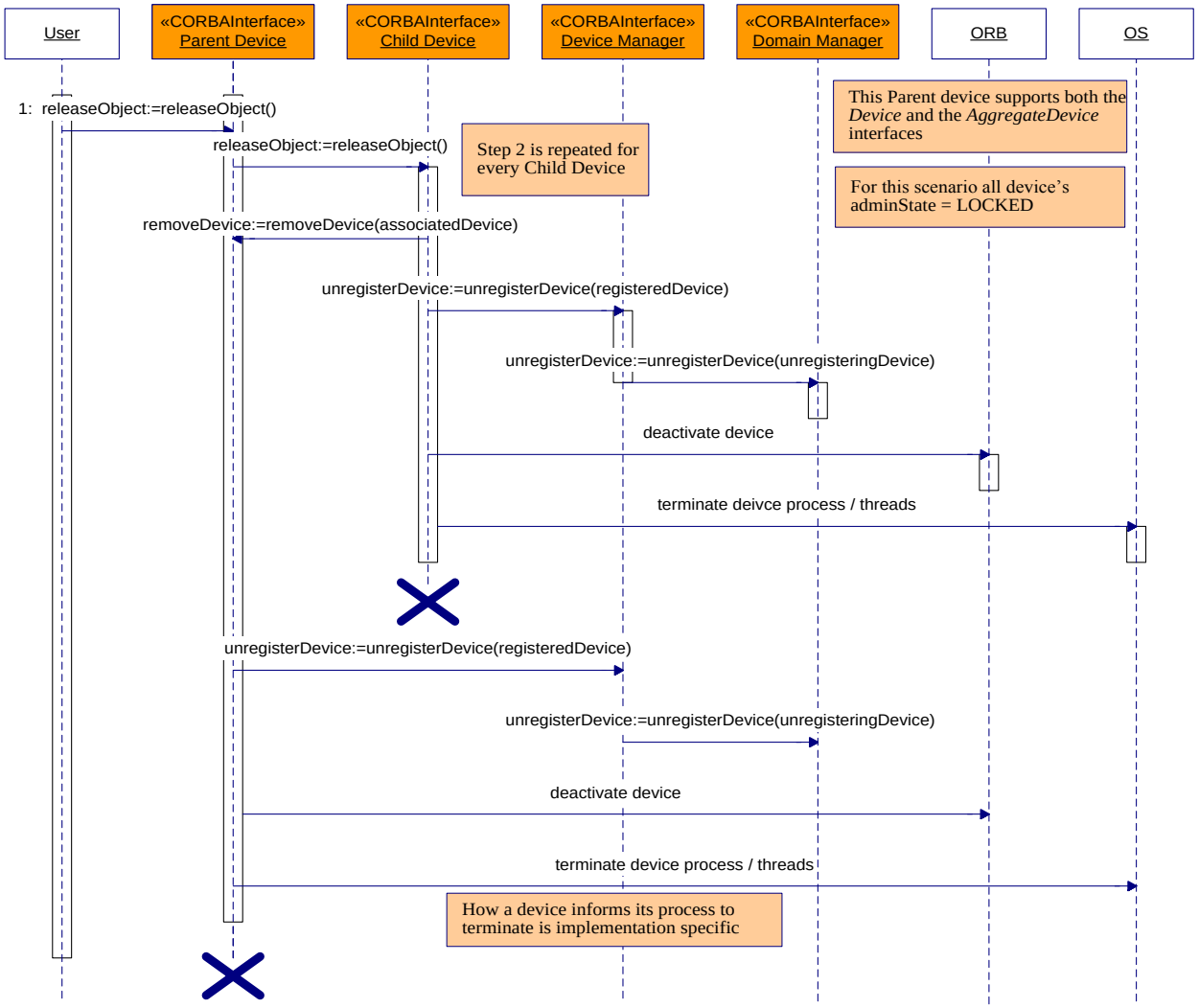

Figure 3-25: Release Composite & Aggregated *Device* Scenario

3.1.3.3.1.5.3.4 Returns

The *releaseObject* operation does not return a value.

3.1.3.3.1.5.3.5 Exceptions/Errors

The *releaseObject* operation shall raise 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.

3.1.3.3.2 *LoadableDevice*

3.1.3.3.2.1 Description

This interface extends the *Device* interface by adding software loading and unloading behavior to a device.

3.1.3.3.2.2 UML

The *LoadableDevice* Interface UML is depicted in Figure 3-26.

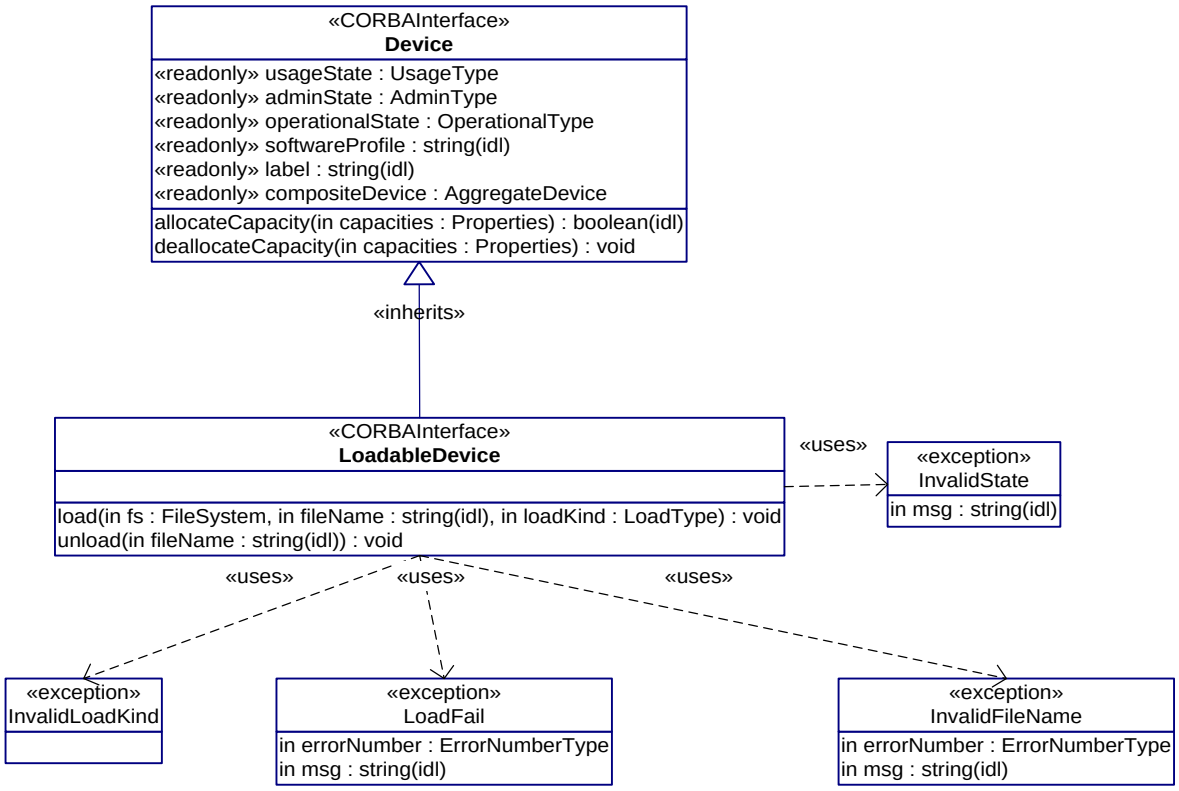

Figure 3-26: *LoadableDevice* Interface UML

3.1.3.3.2.3 Types

3.1.3.3.2.3.1 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.2.1.

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

3.1.3.3.2.3.2 InvalidLoadKind

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

```
exception InvalidLoadKind{};
```

3.1.3.3.2.3.3 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 device. The error number shall indicate a CF ErrorNumberType. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.3.2.4 Attributes

N/A

3.1.3.3.2.5 Operations

3.1.3.3.2.5.1 *load*

3.1.3.3.2.5.1.1 Brief Rationale

The *load* operation provides the mechanism for loading software on a specific device. The loaded software may be subsequently executed on the device, if the device is an executable device.

3.1.3.3.2.5.1.2 Synopsis

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

3.1.3.3.2.5.1.3 Behavior

The *load* operation shall load the file identified by the input filename parameter on the device based upon the input loadKind parameter. The input filename parameter is a pathname relative to the file system identified by the input FileSystem parameter

The *load* operation shall support the load types as stated in the device's software profile LoadType allocation properties.

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.

3.1.3.3.2.5.1.4 Returns

This operation does not return any value.

3.1.3.3.2.5.1.5 Exceptions/Errors

The *load* operation shall raise the InvalidState exception if upon entry the *Device's* adminState attribute is either LOCKED or SHUTTING_DOWN or its operationalState attribute is DISABLED.

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

The *load* operation shall raise the CF InvalidFileName exception when the file designated by the input filename parameter cannot be found.

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

3.1.3.3.2.5.2 *unload*

3.1.3.3.2.5.2.1 Brief Rationale

The *unload* operation provides the mechanism to unload software that is currently loaded.

3.1.3.3.2.5.2.2 Synopsis

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

3.1.3.3.2.5.2.3 Behavior

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

3.1.3.3.2.5.2.4 Returns

This operation does not return a value.

3.1.3.3.2.5.2.5 Exceptions/Errors

The *unload* operation shall raise the InvalidState exception if upon entry the device's adminState attribute is LOCKED or its operationalState attribute is DISABLED.

The *unload* operation shall raise the CF InvalidFileName exception when the file designated by the input filename parameter cannot be found.

3.1.3.3.3 *ExecutableDevice*

3.1.3.3.3.1 Description

This interface extends the *LoadableDevice* interface by adding execute and terminate behavior to a device.

3.1.3.3.3.2 UML

The *ExecutableDevice* Interface UML is depicted in Figure 3-27.

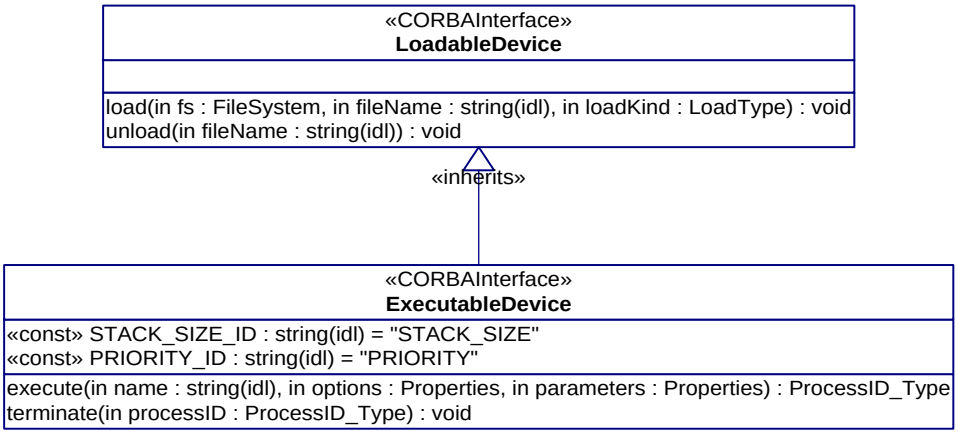

Figure 3-27: *ExecutableDevice* Interface UML

3.1.3.3.3.3 Types

3.1.3.3.3.3.1 InvalidProcess

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

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

3.1.3.3.3.3.2 InvalidFunction

The InvalidFunction exception indicates that a function, as identified by the input name parameter, hasn't been loaded on this device.

```
exception InvalidFunction{};
```

3.1.3.3.3.3.3 ProcessID_Type

The ProcessID_Type defines a process number within the system. The process number is unique to the Processor operating system that created the process.

```
typedef long ProcessID_Type;
```

3.1.3.3.3.3.4 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.

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

3.1.3.3.3.3.5 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.

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

3.1.3.3.3.3.6 STACK_SIZE_ID

The STACK_SIZE_ID is the identifier for the *ExecutableDevice::execute* operation options parameter. The value for a stack size shall be an unsigned long.

```
Constant string STACK_SIZE_ID = "STACK_SIZE";
```

3.1.3.3.3.3.7 PRIORITY_ID

The PRIORITY_ID is the identifier for the *ExecutableDevice::execute* operation options parameters. The value for a priority shall be an unsigned long.

```
Constant string PRIORITY_ID = "PRIORITY";
```

3.1.3.3.3.3.8 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 execute function on the device. The error number shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.3.3.4 Attributes

N/A.

3.1.3.3.3.5 Operations

3.1.3.3.3.5.1 *execute*

3.1.3.3.3.5.1.1 Brief Rationale

The *execute* operation provides the mechanism for starting up and executing a software process/thread on a device.

3.1.3.3.3.5.1.2 Synopsis

```
ProcessID_Type execute (in string name, in Properties options,
in Properties parameters) raises (InvalidState, InvalidFunction,
InvalidParameters, InvalidOptions, InvalidFileName,
ExecuteFail);
```

3.1.3.3.3.5.1.3 Behavior

The *execute* operation shall execute the function or file identified by the input name parameter using the input parameters and options parameters. Whether the input name parameter is a function or a file name is device-implementation-specific.

The *execute* operation shall convert the input parameters (id/value string pairs) parameter to the standard argv of the POSIX exec family of functions, where argv(0) is the function name. The *execute* operation shall map the input parameters parameter to argv starting at index 1 as follows, argv (1) maps to input parameters (0) id and argv (2) maps to input parameters (0) value and so forth. The *execute* operation passes argv through the operating system "execute" function.

The *execute* operation input options parameters are STACK_SIZE_ID and PRIORITY_ID. The *execute* operation shall use these options, when specified, to set the operating system's process/thread stack size and priority, for the executable image of the given input name parameter.

3.1.3.3.3.5.1.4 Returns

The *execute* operation shall return a unique process ID for the process that it created.

3.1.3.3.3.5.1.5 Exceptions/Errors

The *execute* operation shall raise the InvalidState exception if upon entry the device's adminState attribute is either LOCKED or SHUTTING_DOWN or its operationalState attribute is DISABLED.

The *execute* operation shall raise the InvalidFunction exception when the function indicated by the input name parameter does not exist for the device.

The *execute* operation shall raise the CF InvalidFileName exception when the file name indicated by the input name parameter does not exist for the device.

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

The *execute* operation shall raise the InvalidOptions exception when the input options parameter does not comply with sections 3.1.3.3.3.3.6 STACK_SIZE_ID and 3.1.3.3.3.3.7 PRIORITY_ID.

The *execute* operation shall raise the ExecuteFail exception when the operating system "execute" function for the device is not successful.

3.1.3.3.3.5.2 *terminate*

3.1.3.3.3.5.2.1 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.

3.1.3.3.3.5.2.2 Synopsis

```
void terminate (in ProcessID_Type processId) raise
(InvalidProcess, InvalidState);
```

3.1.3.3.3.5.2.3 Behavior

The *terminate* operation shall terminate the execution of the process/thread designated by the processId input parameter on the device.

3.1.3.3.3.5.2.4 Returns

This operation does not return a value.

3.1.3.3.3.5.2.5 Exceptions/Errors

The *terminate* operation shall raise the InvalidState exception if upon entry the device's adminState attribute is LOCKED or its operationalState attribute is DISABLED.

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

3.1.3.3.4 *AggregateDevice*

3.1.3.3.4.1 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.

3.1.3.3.4.2 UML

The *AggregateDevice* Interface UML is depicted in Figure 3-28.

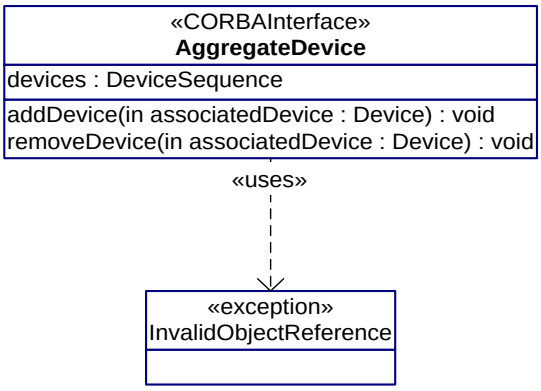

Figure 3-28: *AggregateDevice* Interface UML

3.1.3.3.4.3 Types

N/A.

3.1.3.3.4.4 Attributes

3.1.3.3.4.4.1 devices

The readonly devices attribute shall contain 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.

```
readonly attribute DeviceSequence devices;
```

3.1.3.3.4.5 Operations

3.1.3.3.4.5.1 *addDevice*

3.1.3.3.4.5.1.1 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.

3.1.3.3.4.5.1.2 Synopsis

```
void addDevice (in Device associatedDevice) raises
(InvalidObjectReference);
```

3.1.3.3.4.5.1.3 Behavior

The *addDevice* operation shall add the input associatedDevice parameter to the *AggregateDevice's* devices attribute when the associatedDevice does not exist in the devices attribute. The associatedDevice is ignored when duplicated.

The *addDevice* operation shall write a FAILURE_ALARM log record, upon unsuccessful adding of an associatedDevice to the *AggregateDevice's* devices attribute.

3.1.3.3.4.5.1.4 *Returns*

This operation does not return any *value*.

3.1.3.3.4.5.1.5 Exceptions/Errors

The *addDevice* operation shall raise the CF InvalidObjectReference when the input associatedDevice parameter is a nil CORBA object reference.

3.1.3.3.4.5.2 *removeDevice*

3.1.3.3.4.5.2.1 Brief Rationale

The *removeDevice* operation provides the mechanism to disassociate a device from another device.

3.1.3.3.4.5.2.2 Synopsis

```
void removeDevice (in Device associatedDevice) raises
(InvalidObjectReference);
```

3.1.3.3.4.5.2.3 Behavior

The *removeDevice* operation shall remove the input associatedDevice parameter from the *AggregateDevice's* devices attribute.

The *removeDevice* operation shall write a FAILURE_ALARM log record, upon unsuccessful removal of the associatedDevice from the *AggregateDevice* devices attribute.

3.1.3.3.4.5.2.4 Returns

This operation does not return any value.

3.1.3.3.4.5.2.5 Exceptions/Errors

The *removeDevice* operation shall raise the CF InvalidObjectReference when the input associatedDevice parameter is a nil CORBA object reference or does not exist in the *AggregateDevice* devices attribute.

3.1.3.4 Framework Services Interfaces

Framework Services Interfaces shall be implemented using the CF IDL presented in Appendix C.

3.1.3.4.1 *File*

3.1.3.4.1.1 Description

The *File* interface provides the ability to read and write files residing within a compliant, distributed file system. A file can be thought of conceptually as a sequence of octets with a current filePointer describing where the next read or write will occur. This filePointer points to the beginning of the file upon construction of the file object. The *File* interface is modeled after the POSIX/C file interface.

3.1.3.4.1.2 UML

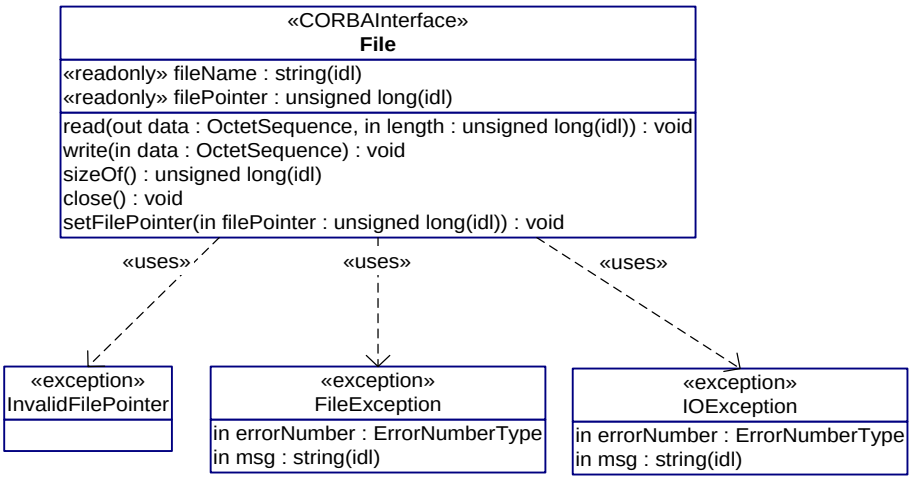

Figure 3-29: *File* Interface UML

3.1.3.4.1.3 Types

3.1.3.4.1.3.1 IOException

The IOException exception indicates an error occurred during a *read* or *write* operation to a file. The error number shall indicate a CF ErrorNumberType value. The message is component-dependent, providing additional information describing the reason for the error.

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

3.1.3.4.1.3.2 InvalidFilePointer

The InvalidFilePointer exception indicates the file pointer is out of range based upon the current file size.

```
exception InvalidFilePointer{};
```

3.1.3.4.1.4 Attributes

3.1.3.4.1.4.1 fileName

The readonly fileName attribute shall contain the pathname used as the input fileName parameter of the *FileSystem::create* operation when the file was created .

```
readonly attribute string fileName;
```

3.1.3.4.1.4.2 filePointer

The readonly filePointer attribute shall contain the current file position. The filePointer attribute value dictates where the next read or write will occur.

```
readonly attribute unsigned long filePointer;
```

3.1.3.4.1.5 Operations

3.1.3.4.1.5.1 *read*

3.1.3.4.1.5.1.1 Brief Rationale

Applications require the *read* operation in order to retrieve data from remote files.

3.1.3.4.1.5.1.2 Synopsis

```
void read (out OctetSequence data, in unsigned long length)
raises (IOException);
```

3.1.3.4.1.5.1.3 Behavior

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. The *read* operation shall read less than the number of octets specified in the input-length parameter, when an end of file is encountered.

3.1.3.4.1.5.1.4 Returns

The *read* operation shall return via the out Message parameter a CF OctetSequence that equals the number of octets actually read from the *File*. If the filePointer attribute value reflects the end of the *File*, the *read* operation shall return a zero-length CF OctetSequence.

3.1.3.4.1.5.1.5 Exceptions/Errors

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

3.1.3.4.1.5.2 *write*

3.1.3.4.1.5.2.1 Brief Rationale

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

3.1.3.4.1.5.2.2 Synopsis

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

3.1.3.4.1.5.2.3 Behavior

The *write* operation shall write data to the file referenced. The *write* operation shall increment the filePointer attribute to reflect the number of octets written, when the operation is successful. If the *write* 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.

3.1.3.4.1.5.2.4 Returns

This operation does not return any value.

3.1.3.4.1.5.2.5 Exceptions/Errors

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

3.1.3.4.1.5.3 sizeOf

3.1.3.4.1.5.3.1 Brief Rationale

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

3.1.3.4.1.5.3.2 Synopsis

```
unsigned long sizeOf() raises (FileException);
```

3.1.3.4.1.5.3.3 Behavior

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

3.1.3.4.1.5.3.4 Returns

The *sizeOf* operation shall return the number of octets stored in the file.

3.1.3.4.1.5.3.5 Exceptions/Errors

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

3.1.3.4.1.5.4 *close*

3.1.3.4.1.5.4.1 Brief Rationale

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

3.1.3.4.1.5.4.2 Synopsis

```
void close() raises (FileException);
```

3.1.3.4.1.5.4.3 Behavior

The *close* operation shall release any OE file resources associated with the component. The *close* operation shall make the file unavailable to the component.

3.1.3.4.1.5.4.4 Returns

This operation does *not* return any value.

3.1.3.4.1.5.4.5 Exceptions/Errors.

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

3.1.3.4.1.5.5 *setFilePointer*

3.1.3.4.1.5.5.1 Brief Rationale

The *setFilePointer* operation positions the file pointer where the next read or write will occur.

3.1.3.4.1.5.5.2 Synopsis

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

3.1.3.4.1.5.5.3 Behavior

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

3.1.3.4.1.5.5.4 Returns

This operation does not return any value.

3.1.3.4.1.5.5.5 Exceptions/Errors

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.

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

3.1.3.4.2 *FileSystem*

3.1.3.4.2.1 Description

The FileSystem interface defines CORBA operations that enable remote access to a physical file system. (see Figure 3-30)

The files stored on a file system may be plain files or directories. Valid individual filenames and directory names shall be 40 characters or less. 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 are structured according to the POSIX specification whose valid characters include the "/" (forward slash) character in addition to the valid filename characters. A valid pathname may consist of a single filename. A valid pathname shall not exceed 1024 characters.

3.1.3.4.2.2 UML

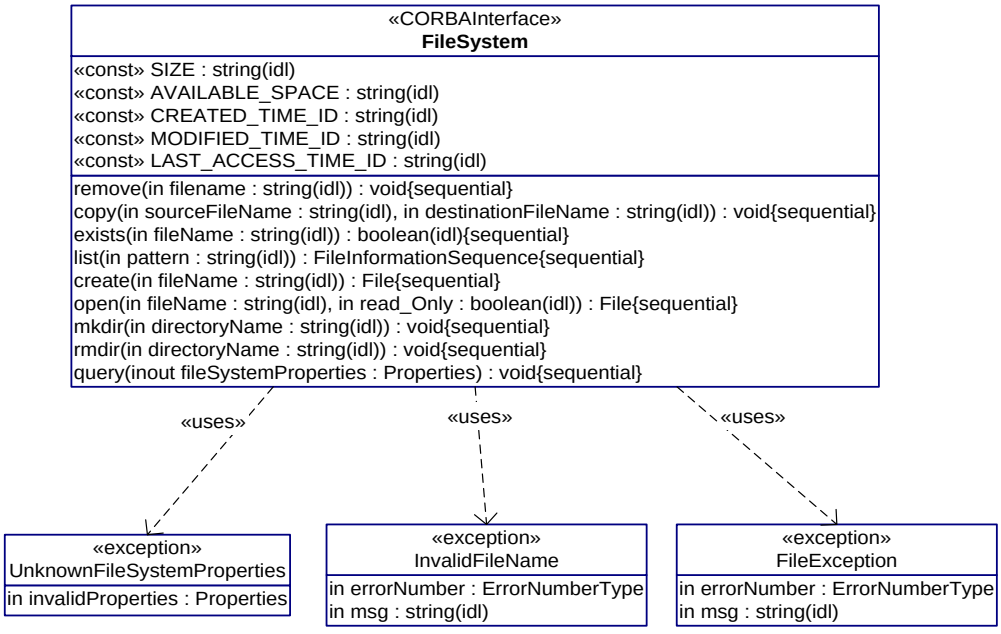

Figure 3-30: *FileSystem* Interface UML

3.1.3.4.2.3 Types

3.1.3.4.2.3.1 UnknownFileSystemProperties.

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

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

3.1.3.4.2.3.2 fileSystemProperties Query Constants

Constants are defined to be used for the *query* operation (see section 3.1.3.4.2.5.9).

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

3.1.3.4.2.3.3 FileInformationType

The FileInformationType indicates the information returned for a file. Not all the fields in the FileInformationType are applicable for all file systems. At a minimum, the file system 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.

```
struct FileInformationType
{
     string              name;
     FileType            kind;
     unsigned long long  size;
     Properties          fileProperties;
};
```

The *name* element of the FileInformationType indicates the simple name of the file. The *kind* element of the FileInformationType indicates the type of the file entry. The *size* element of the FileInformationType indicates the size in octets.

3.1.3.4.2.3.4 FileInformationSequence

The FileInformationSequence type defines an unbounded sequence of FileInformationTypes.

```
typedef sequence<FileInformationType>FileInformationSequence;
```

3.1.3.4.2.3.5 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 FileSystem.

```
enum FileType
{
     PLAIN,
     DIRECTORY,
     FILE_SYSTEM
};
```

3.1.3.4.2.3.6 CREATED_TIME_ID

The *fileProperies* element of the FileInformationType may be used to indicate the time a file was created. 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.

```
Constant string CREATED_TIME_ID = "CREATED_TIME";
```

3.1.3.4.2.3.7 MODIFIED_TIME_ID

The *fileProperies* element of the FileInformationType may be used to indicate the time a file was last modified. 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.

```
Constant string MODIFIED_TIME_ID="MODIFIED_TIME";
```

3.1.3.4.2.3.8 LAST_ACCESS_TIME_ID

The *fileProperies* element of the FileInformationType may be used to indicate the time a file was last accessed. 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.

```
Constant string LAST_ACCESS_TIME_ID="LAST_ACCESS_TIME";
```

3.1.3.4.2.4 Attributes

N/A.

3.1.3.4.2.5 Operations

3.1.3.4.2.5.1 *remove*

3.1.3.4.2.5.1.1 Brief Rationale

The *remove* operation provides the ability to remove a plain file from a file system.

3.1.3.4.2.5.1.2 Synopsis

```
void remove (in string fileName) raises (FileException,
InvalidFileName);
```

3.1.3.4.2.5.1.3 Behavior

The *remove* operation shall remove the plain file which corresponds to the input fileName parameter.

3.1.3.4.2.5.1.4 Returns

This operation does not return any value.

3.1.3.4.2.5.1.5 Exceptions/Errors

The *remove* operation shall raise the CF InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.

The *remove* operation shall raise the CF FileException when a file-related error occurs.

3.1.3.4.2.5.2 *copy*

3.1.3.4.2.5.2.1 Brief Rationale

The *copy* operation provides the ability to copy a plain file to another plain file.

3.1.3.4.2.5.2.2 Synopsis

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

3.1.3.4.2.5.2.3 Behavior

The *copy* operation shall copy the source file identified by the input sourceFileName parameter to the destination file identified by the input destinationFileName parameter.

The *copy* operation shall overwrite the destination file, when the destination file already exists and is not identical to the source file.

3.1.3.4.2.5.2.4 Returns

This operation does not return any value.

3.1.3.4.2.5.2.5 Exceptions/Errors

The *copy* operation shall raise the CF FileException exception when a file-related error occurs.

The copy operation shall raise the CF InvalidFileName exception when the destination pathname is identical to the source pathname.

The *copy* operation shall raise the CF InvalidFileName exception when the sourceFileName or destinationFileName input parameters are not a valid absolute pathnames.

3.1.3.4.2.5.3 *exists*

3.1.3.4.2.5.3.1 Brief Rationale

The *exists* operation provides the ability to verify the existence of a file within a file system.

3.1.3.4.2.5.3.2 Synopsis

```
boolean exists (in string fileName) raises (InvalidFileName);
```

3.1.3.4.2.5.3.3 Behavior

The *exists* operation shall check to see if a file exists based on the fileName parameter.

3.1.3.4.2.5.3.4 Returns

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

3.1.3.4.2.5.3.5 Exceptions/Errors

The *exists* operation shall raise the CF InvalidFileName exception when input fileName parameter is not a valid absolute pathname.

3.1.3.4.2.5.4 *list*

3.1.3.4.2.5.4.1 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.

3.1.3.4.2.5.4.2 Synopsis

```
FileInformationSequence list (in string pattern) raises
(FileException, InvalidFileName);
```

3.1.3.4.2.5.4.3 Behavior

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

3.1.3.4.2.5.4.4 Returns

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

3.1.3.4.2.5.4.5 Exceptions/Errors

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.

The *list* operation shall raise the CF FileException when a file-related error occurs.

3.1.3.4.2.5.5 *create*

3.1.3.4.2.5.5.1 Brief Rationale

The *create* operation provides the ability to create a new plain file on the file system.

3.1.3.4.2.5.5.2 Synopsis

```
File create (in string fileName) raises (InvalidFileName,
FileException);
```

3.1.3.4.2.5.5.3 Behavior

The *create* operation shall create a new *File* based upon the input fileName parameter.

3.1.3.4.2.5.5.4 Returns

The *create* operation shall return a file object reference to the opened file.

3.1.3.4.2.5.5.5 Exceptions/Errors

The *create* operation shall raise the CF FileException if the file already exists or another file error occurred.

The *create* operation shall raise the CF InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.

3.1.3.4.2.5.6 *open*

3.1.3.4.2.5.6.1 Brief Rationale

The *open* operation provides the ability to open a plain file for read or write.

3.1.3.4.2.5.6.2 Synopsis

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

3.1.3.4.2.5.6.3 Behavior

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

3.1.3.4.2.5.6.4 Returns

The *open* operation shall return a *File* instance on successful completion. The *open* operation shall set the filePointer attribute of the returned file instance to the beginning of the file.

3.1.3.4.2.5.6.5 Exceptions/Errors

The *open* operation shall raise the CF FileException if the file does not exist or another file error occurred.

The *open* operation shall raise the CF InvalidFileName exception when the input fileName parameter is not a valid absolute pathname.

3.1.3.4.2.5.7 *mkdir*

3.1.3.4.2.5.7.1 Brief Rationale

The *mkdir* operation provides the ability to create a directory on the file system.

3.1.3.4.2.5.7.2 Synopsis

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

3.1.3.4.2.5.7.3 Behavior

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

3.1.3.4.2.5.7.4 Returns.

This operation does not return any value.

3.1.3.4.2.5.7.5 Exceptions/Errors

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.

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

3.1.3.4.2.5.8 *rmdir*.

3.1.3.4.2.5.8.1 Brief Rationale

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

3.1.3.4.2.5.8.2 Synopsis

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

3.1.3.4.2.5.8.3 Behavior

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

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

3.1.3.4.2.5.8.4 Returns

This operation does not return any value.

3.1.3.4.2.5.8.5 Exceptions/Errors

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.

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

3.1.3.4.2.5.9 *query*

3.1.3.4.2.5.9.1 Brief Rationale

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

3.1.3.4.2.5.9.2 Synopsis

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

3.1.3.4.2.5.9.3 Behavior

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

The *FileSystem::query* operation shall recognize and provide the designated return values for the following *fileSystemProperties* (section 3.1.3.4.2.3.2):

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 3.1.3.4.2.3.2 for the constants for the fileSystemProperties.

3.1.3.4.2.5.9.4 Returns

This operation does not return any value.

3.1.3.4.2.5.9.5 Exceptions/Errors

The *query* operation shall raise the UnknownFileSystemProperties exception when the given file system property is not recognized.

3.1.3.4.3 *FileManager*

3.1.3.4.3.1 Description

Multiple, distributed file systems may be accessed through a file manager. The *FileManager* interface appears to be a single file system although the actual file storage may span multiple physical file systems. (Reference the *FileManager* interface UML in Figure 3-31.)

This is called a federated file system. A federated file system is created 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*. Based upon the pathname of a directory or file and the set of mounted file systems, the file manager delegates the *FileSystem* 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 *FileSystem'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 file manager delegates the operation to the appropriate file systems (based upon supplied pathnames) thereby allowing copy of files between file systems.

If a client does not need to mount and unmount file systems, it may treat the file manager as a file system by CORBA widening a *FileManager* reference to a *FileSystem* reference. One can always widen a *FileManager* to a *FileSystem* since the *FileManager* is derived from a *FileSystem*.

3.1.3.4.3.2 UML

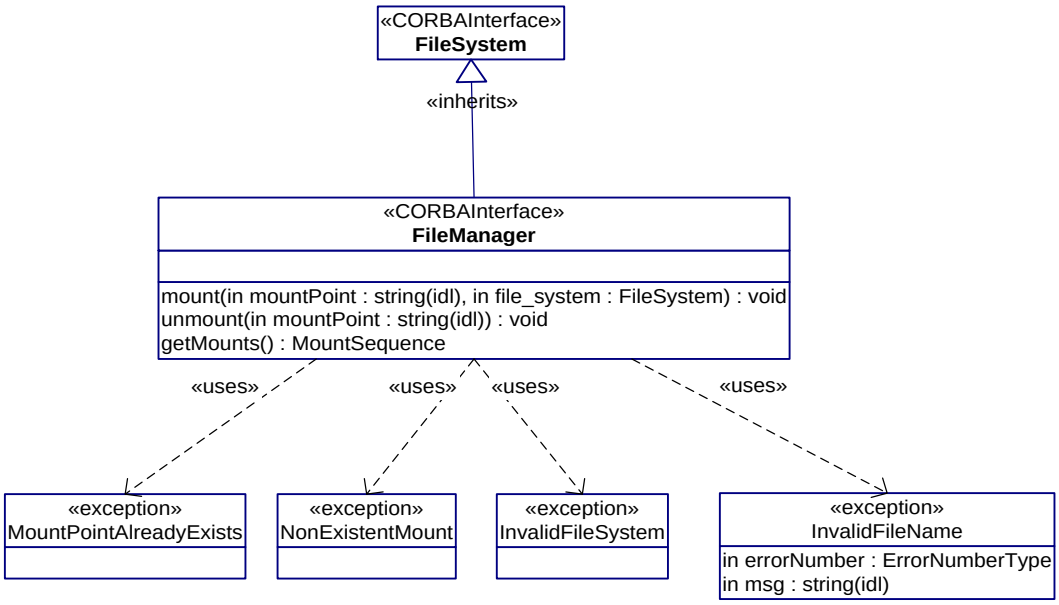

Figure 3-31: *FileManager* Interface UML

3.1.3.4.3.3 Types

3.1.3.4.3.3.1 MountType

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

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

3.1.3.4.3.3.2 MountSequence

The MountSequence is an unbounded sequence of Mount types.

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

3.1.3.4.3.3.3 NonExistentMount

The NonExistentMount exception indicates a mount point does not exist within the *FileManager*.

```
exception NonExistentMount{};
```

3.1.3.4.3.3.4 MountPointAlreadyExists

The MountPointAlreadyExists exception indicates the mount point is already in use in the *FileManager*.

```
exception MountPointAlreadyExists{};
```

3.1.3.4.3.3.5 InvalidFileSystem

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

```
exception InvalidFileSystem{};
```

3.1.3.4.3.4 Attributes

N/A

3.1.3.4.3.5 Operations

3.1.3.4.3.5.1 *mount*

3.1.3.4.3.5.1.1 Brief Rationale

The file manager 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).

3.1.3.4.3.5.1.2 Synopsis

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

3.1.3.4.3.5.1.3 Behavior

The *mount* operation shall associate the specified file system with the mount point referenced by the input mountPoint parameter. A mount point name shall begin with a "/" (forward slash character). The input mountPoint parameter is a logical directory name for a file system.

3.1.3.4.3.5.1.4 Returns.

This operation does not return any value.

3.1.3.4.3.5.1.5 Exceptions/Errors.

The *mount* operation shall raise the CF InvalidFileName exception when the input mount point does not conform to the file name syntax in section 3.1.3.4.2.1.

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

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

3.1.3.4.3.5.2 *unmount*

3.1.3.4.3.5.2.1 Brief Rationale

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

3.1.3.4.3.5.2.2 Synopsis

```
void unmount (in string mountPoint) raises (NonExistentMount);
```

3.1.3.4.3.5.2.3 Behavior

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

3.1.3.4.3.5.2.4 Returns

This operation does not return any value.

3.1.3.4.3.5.2.5 Exceptions/Errors

The *unmount* operation shall raise the NonExistentMount exception when the mount point does not exist.

3.1.3.4.3.5.3 *getMounts*

3.1.3.4.3.5.3.1 Brief Rationale

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

3.1.3.4.3.5.3.2 Synopsis

```
MountSequence getMounts();
```

3.1.3.4.3.5.3.3 Behavior

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

3.1.3.4.3.5.3.4 Returns

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

3.1.3.4.3.5.3.5 Exceptions/Errors

This operation does not raise any exceptions.

3.1.3.4.3.5.4 File System Operations.

The system may support multiple *FileSystem* implementations. Some file systems correspond directly to a physical file system within the system. The *FileManager* interface shall support a federated, or distributed, file system that may span multiple *FileSystem* components. From the client perspective, the *FileManager* may be used just like any other *FileSystem* component since the *FileManager* inherits all the *FileSystem* operations.

A file manager shall implement the inherited *FileSystem* operations as required under section 3.1.3.4.2 for each mounted file system. The *FileSystem* operations ensure that the filename/directory arguments given are absolute pathnames relative to a mounted file system. The *FileSystem* operations inherited by a file manager shall remove the name of the mounted file system from input pathnames before passing the pathnames to any operation on a mounted file system. The file manager shall propagate exceptions raised by a mounted file system.

The file manager shall use the *FileSystem* operations of the file system whose associated mount point exactly matches the input fileName parameter to the lowest matching subdirectory.

3.1.3.4.3.5.5 *query*

3.1.3.4.3.5.5.1 Brief Rationale

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

3.1.3.4.3.5.5.2 Synopsis

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

3.1.3.4.3.5.5.3 Behavior

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

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.

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.

3.1.3.4.3.5.5.4 Returns

This operation does not return any value.

3.1.3.4.3.5.5.5 Exceptions/Errors

The *query* operation shall raise the UnknownFileSystemProperties exception when the input fileSystemProperties parameter contains an invalid property ID element

3.1.3.5 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 XML vocabulary.

The types of XML files that are used to describe a system's hardware and software assets are depicted in Figure 3-32. The XML vocabulary within each of these files describes a distinct aspect of the hardware and software assets. The collection of XML 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 Software Package Descriptor – all profiles for CORBA components contain a Software Component Descriptor. A software profile for an application contains a Software Assembly descriptor (3.1.3.2.1.4.1), the device manager profile contains a Device Configuration Descriptor (3.1.3.2.4.4.4), and the domain manager software profile contains a DomainManager Configuration Descriptor (3.1.3.2.3.4.5).

Domain Profile files shall be complaint to the Document Type Definitions (DTDs) provided in Appendix D. DTD files are installed in the domain and shall have ".dtd" as their filename extension. All XML files shall have as the first two lines as an XML declaration (?xml) and a document type declaration (!DOCTYPE). The XML declaration specifies the XML version and whether the document is standalone. The document type declaration specifies the DTD for the document. Example declarations are as follows:

<?xml version=”1.0” standalone=”no”?>

<!DOCTYPE softwareassembly SYSTEM “softwareassembly.2.2.2.dtd”>

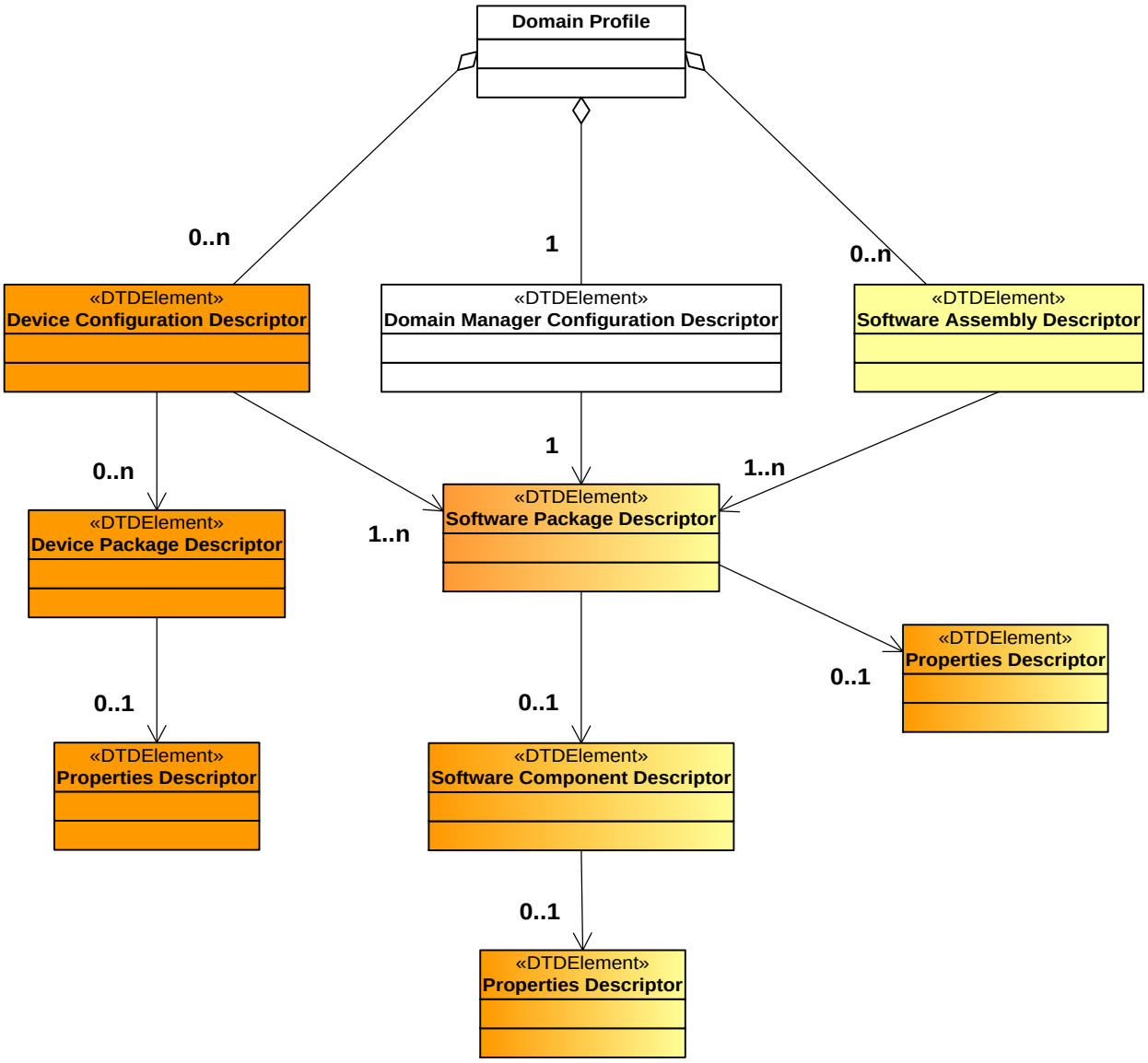

Figure 3-32: Relationship of Domain Profile XML File Types

3.1.3.5.1 Software Package Descriptor

A Software Package Descriptor (SPD) identifies a software component implementation(s). A Software Package Descriptor file shall have a “.spd.xml” extension. 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 a Software Package Descriptor file.

3.1.3.5.2 Software Component Descriptor

A Software Component Descriptor (SCD) contains information about a specific SCA software component (*Resource, ResourceFactory, Device*). A Software Component Descriptor file shall

have a ".scd.xml" extension. A Software Component Descriptor file contains information about the interfaces that a component provides and/or uses. A Software Component Descriptor for a *Device* type has a reference to Device Package Descriptor file.

3.1.3.5.3 Software Assembly Descriptor

A Software Assembly Descriptor (SAD) contains information about the components that make up an application. The application factory uses this information when creating an application. A Software Assembly Descriptor file shall have a ".sad.xml" extension.

3.1.3.5.4 Properties Descriptor

A Property File contains information about the properties applicable to a software package or a device package. A Properties File shall have a ".prf.xml" extension. A Properties File contains information about the properties of a component such as configuration, test, execute, and allocation types.

3.1.3.5.5 Device Package Descriptor

A Device Package Descriptor (DPD) identifies a class of a device. A Device Package Descriptor File shall have a ".dpd.xml" extension. A Device Package Descriptor also has Properties that define specific properties (capacity, serial number, etc.) for this class of device.

3.1.3.5.6 Device Configuration Descriptor

A Device Configuration Descriptor (DCD) contains information about the devices associated with a device manager, how to find the domain manager, and the configuration information (*Log*, *FileSystems*, etc.) for a device. A Device Configuration Descriptor file shall have a ".dcd.xml" extension.

3.1.3.5.7 Profile Descriptor

A Profile Descriptor is an XML element which contains an absolute pathname for a Software Package Descriptor (SPD), Software Assembly Descriptor (SAD), DomainManager Configuration Descriptor (DMD), or a Device Configuration Descriptor (DCD), depending upon the context. This element is used as the parameter for interface profile attributes (e.g., CF Application, CF Device, CF ApplicationFactory, CF DeviceManager, CF DomainManager).

3.1.3.5.8 DomainManager Configuration Descriptor

A DomainManager Configuration Descriptor (DMD) contains configuration information for the domain manager. A DomainManager Configuration Descriptor file shall have a ".dmd.xml" extension.

3.1.3.6 Core Framework Base Types

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

3.1.3.6.1 DataType

This type is a CORBA IDL structure, which may be used to hold any CORBA basic type or static IDL type. The id attribute indicates the kind of value and type (e.g., frequency, preset, etc.). The id may be an UUID string, an integer string, or a name identifier depending on context. The value attribute may be any static IDL type or CORBA basic type.

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

3.1.3.6.2 DeviceSequence

The CF DeviceSequence type defines an unbounded sequence of devices.

```
typedef sequence <Device> DeviceSequence;
```

3.1.3.6.3 FileException

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

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

3.1.3.6.4 InvalidFileName

The CF InvalidFileName exception indicates an invalid file name was passed to a file service operation. The error number shall indicate a CF ErrorNumberType value. The message provides information describing why the filename was invalid.

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

3.1.3.6.5 InvalidObjectReference

The CF InvalidObjectReference exception indicates an invalid CORBA object reference error.

```
exception InvalidObjectReference {string msg;};
```

3.1.3.6.6 InvalidProfile

The CF InvalidProfile exception indicates an invalid profile error.

```
exception InvalidProfile{};
```

3.1.3.6.7 OctetSequence

This type is a CORBA unbounded sequence of octets.

```
typedef  sequence <octet> OctetSequence;
```

3.1.3.6.8 Properties

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

```
typedef sequence <DataType> Properties;
```

3.1.3.6.9 StringSequence

This type defines a sequence of strings.

```
typedef sequence <string> StringSequence;
```

3.1.3.6.10 UnknownProperties

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

```
exception UnknownProperties {Properties invalidProperties; };
```

3.1.3.6.11 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;
};
```

3.1.3.6.12 DeviceAssignmentSequence

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

```
typedef sequence <DeviceAssignmentType>
DeviceAssignmentSequence;
```

3.1.3.6.13 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 [4].

CF_NOTSET 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.)

```
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_EFBIG, CF_EINPROGRESS,
CF_EINTR,CF_EINVAL, CF_EIO, CF_EISDIR, CF_EMFILE, CF_EMLINK,
CF_EMSGSIZE, 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_EPIPE, CF_ERANGE , CF_EROFS, CF_ESPIPE, CF_ESRCH,
CF_ETIMEDOUT ,CF_EXDEV
};
```

3.2 APPLICATIONS

Applications are programs that perform the functions of a specific SCA-compliant product. 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.

3.2.1 General Application Requirements

An application's dependencies to the log, file manager, file system, CORBA Event Service, and CORBA Naming Service are specified as connections in the SAD using the *domainfinder* element.

3.2.1.1 OS Services

Applications shall be limited to using the OS services that are designated as mandatory in the SCA Application Environment Profile (Appendix B).

Applications shall perform file access through the CF *File* interfaces. The application filename syntax is specified in section 3.1.3.4.2.1.

All application processes shall have a handler registered for the POSIX-defined SIGQUIT signal.

3.2.1.2 CORBA Services

Applications shall be limited to using CORBA and CORBA services defined in the referenced minimumCORBA specification [5]. Dynamically-created stringified IORs may be used to provide an IOR reference value parameter. Applications shall not utilize static stringified IORs.

Applications may support the *LogProducer* interface of the CORBA Lightweight Log Specification [7].

3.2.1.3 CF Interfaces

Applications shall implement the Base Application Interfaces as specified in section 3.1.3.1 using the corresponding IDL in Appendix C. Use of the *ResourceFactory* interface per section 3.1.3.1.7 is optional.

Each application component shall support the mandatory Naming Context IOR, Name Binding, and the identifier execute parameters as described in 3.1.3.2.2.5.1, in addition to their user-defined execute properties in the component's SPD. Each application component shall bind its object reference to the Naming Context IOR using the Name Binding parameter. Each executable component of an application shall set its identifier attribute using the component identifier execute parameter.

Each executable component of an application shall accept the standard argv arguments of the POSIX exec family of functions [4].

An application, each application component, and each device manager shall be accompanied by the appropriate Domain Profile files per section 3.1.3.5.

3.2.2 Application Interfaces

Applications consist of one to many components. These components may be CORBA-capable or not CORBA-capable components. For CORBA-capable components, in addition to supporting the CF Base Application interfaces, the component may implement and use component-specific interfaces for data and/or control. Interfaces provided by a component shall

be described in a Software Component Descriptor file as provides ports. Interfaces required by a component shall be described in a Software Component Descriptor file as uses ports.

An application may define interfaces that are visible to entities external to the application. These external interfaces are *Ports*, referenced in the application SAD *externalports* element. An application interface shall be referenced in the application's SAD *externalports* element, and thus declared "external", if the interface provides a service that is used by other applications.

All non-standard interfaces shall be defined in Interface Control Documents that are available to other parties without restriction to the extent that interfacing or replacement hardware and software can be developed by other parties without restriction.

3.2.2.1 Service Definitions

SCA service definitions consist of APIs, behavior, state, priority and additional information that provide the contract between the Service Provider and the Service User. IDL is used to define the interfaces for service definitions to foster reuse and interoperability. IDL provides a method to inherit from multiple interfaces to form a new service definition.

All SCA APIs shall have their interfaces described in IDL. All non-IDL interfaces shall provide an IDL mapping within the service definition.

3.3 LOGICAL DEVICE

A logical device is a software component that implements one of the Base Device Interfaces. The Base Device Interfaces are *Device, LoadableDevice, ExecutableDevice, and AggregateDevice* as stated in section 2.2.2. and depicted in Figure 3-33.

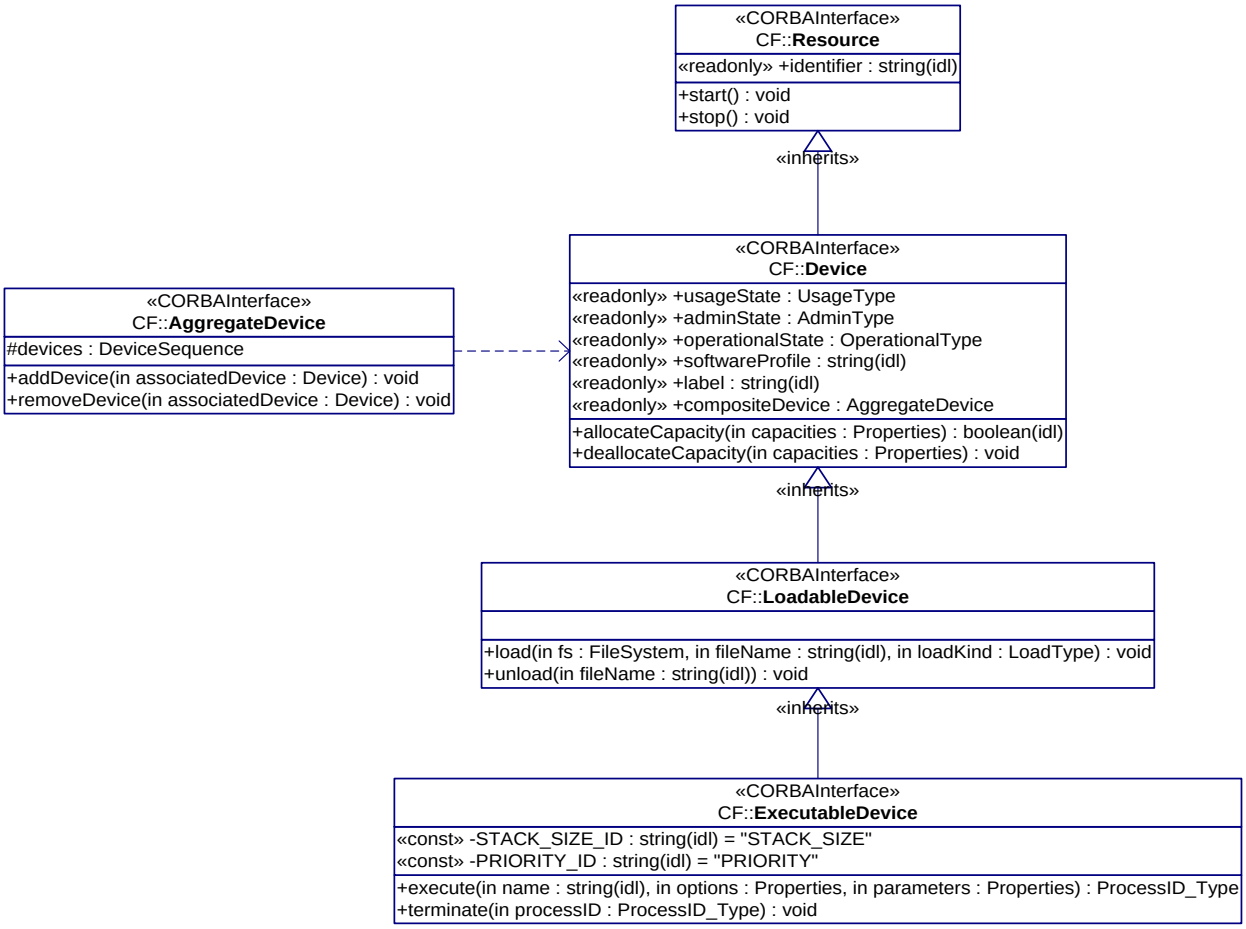

Figure 3-33: Logical *Device* Interface Relationships

3.3.1 OS Services

Logical devices may use any service provided by the OE and as such are not restricted to using the services specified in the SCA Application Environment Profile (Appendix B).

The executable parameters of a logical device shall accept the standard argv arguments as used in the POSIX exec family of functions [4].

A logical device shall accept the executable parameters as specified in section 3.1.3.3.3.5.1.3 (*ExecutableDevice::execute*).

3.3.2 CORBA Services.

Logical devices shall be limited to using CORBA and CORBA services defined in the referenced minimumCORBA specification [5].

Logical devices may support the *LogProducer* interface of the CORBA Lightweight Log Specification [7].

3.3.3 CF Interfaces

A logical device implements one of the following CF interfaces: *Device, LoadableDevice* or *ExecutableDevice*.

In addition to the requirements stated in the *Device* interface (section 3.1.3.3.1), a logical device has the requirements as stated in the *Resource*, *PropertySet*, *Lifecycle*, *Port*, *PortSupplier* and *TestableObject* interfaces.

A logical device shall register itself with a device manager using the value associated with the DEVICE_MGR_IOR parameter per 3.1.3.2.4.5.

A child device shall add itself to a parent device using the executable Composite Device IOR parameter per 3.1.3.2.4.5.

The values associated with the parameters (PROFILE_NAME, COMPOSITE_DEVICE_IOR, DEVICE_ID and DEVICE_LABEL) as described in 3.1.3.2.4.5 shall be used to set the *Device's softwareProfile*, *compositeDevice*, *identifier*, and *label* attributes, respectively.

Hardware critical interfaces shall be defined in Interface Control Documents that are available to other parties without restriction. Critical interfaces are those interfaces at the physical boundary of a replaceable device that are required for the operation and maintenance of the device.

Additional service APIs and their ports beyond the CF adhere to the requirements as described in section 3.2.2.

3.3.4 Profile

Each logical device shall have a SPD, SCD, DPD, and one or more Properties Descriptors as described in section 3.1.3.5. For each logical device, allocation properties shall be defined in its referenced SPD's property file.

3.4 GENERAL SOFTWARE RULES

This section identifies those rules and recommendations specific to the Software Communications Architecture that are not specifically addressed elsewhere in this specification.

3.4.1 Software Development Languages

3.4.1.1 New Software

Software developed for an SCA-compliant system shall be developed in a standard higher order language. The goal of new development should be to provide software that is independent from platform and environment dependencies, ensuring minimal portability issues.

3.4.1.2 Legacy Software

Legacy software is not required to be rewritten in a standard higher order language. Legacy software shall interface with the Core Framework in accordance with this specification.

4 ARCHITECTURE COMPLIANCE

This section defines the authorities as well as the requirements and criteria for the certification of any product to this specification.

Certification may be requested for any product meeting all applicable requirements identified within the scope of the specification. The applicable requirements for any product not fulfilling all requirements of this specification are determined at the sole discretion of the Certification Authority (section 4.1)

This process is based on the existence of three distinct organizations: a Certification Authority (CA), a Specification Authority (SA), and a Test and Evaluation Authority (TA). The CA is given the sole responsibility for granting certification for all products to the specified standard, based on the data and recommendations provided. The Certification Authority is supported by the Specification Authority (SA), which is responsible for developing, maintaining, evolving and interpreting the standard, and the Test and Evaluation Authority (TA) which is responsible for the definition of all test procedures, development and maintenance of all test tools, and for providing formal certification test results.

4.1 CERTIFICATION AUTHORITY

The Joint Program Executive Office (JPEO) JTRS is the Certification Authority (CA) for the SCA and is given the sole responsibility and authority for granting certification of all products to this specification and to certify that a product meets the requirements of this specification. The JPEO JTRS authority is derived from its Charter [C].

4.2 SPECIFICATION AUTHORITY

The Joint Program Executive Office (JPEO) JTRS is the Specification Authority (SA) for the SCA and is given the sole responsibility and authority to incorporate changes, recommendations, additions, or retractions into this specification.

4.3 RESPONSIBILITY FOR COMPLIANCE EVALUATION

The Joint Program Executive Office (JPEO) JTRS shall assign one or more test organizations as the Test and Evaluation Authority (TA) for the SCA. The TA has the responsibility for providing formal certification test results to the Certification Authority.

4.4 EVALUATING COMPLIANCE

Compliance to this specification requires a product to meet all applicable requirements identified within the scope of the specification. Applicability of requirements to specific products is determined by the Certification Authority. Products are submitted to the Test and Evaluation Authority for verification. Results of that verification are submitted to the Certification Authority for evaluation.

The CA grants three levels of product certification for all JTRS standards: Fully Compliant, Compliant with Waivers, and Non-Compliant. A certification of Fully Compliant will be granted when a product has passed all requirements identified by the TA, without exception, for a

specific version of the standard. A product will be certified as Compliant with Waivers when all requirements not validated according to the criteria for a Fully Compliant certification, are granted waivers under the process defined in the JTRS Standards Waiver Process [D]. A product will be declared Non-Compliant when any failed requirement exists for which a waiver is not approved.

4.5 REGISTRATION.

Some elements of an SCA implementation are identified with a Universally Unique Identifier (UUID). As used in this specification, the UUID is defined by the DCE UUID standard adopted by the Common Object Request Broker Architecture (CORBA) [9]. No centralized authority is required to administer UUIDs under this specification.