

SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION

USER'S GUIDE



30 November 2015
Version: 4.1<DRAFT>

Prepared by:

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

Statement A - Approved for public release; distribution is unlimited (30 NOV 15)

REVISION SUMMARY

Version	Revision
0.3	Initial Release
1.0	SCA 4.0 Release
4.1<DRAFT>	Terminology updated and sections added to correspond to SCA 4.1 Release. Version number updated to correspond with SCA release.

TABLE OF CONTENTS

1	SCOPE	8
1.1	Informative References	8
2	SCA INTRODUCTION	9
2.1	Separation of Waveform and Operating Environment.....	9
2.2	Operating Environment	9
2.2.1	Application Environment Profiles	9
2.2.2	Middleware and Data Transfer	10
2.3	JTNC Application Program Interfaces.....	10
3	TOPIC ORIENTED GUIDANCE AND SUPPLEMENTARY INFORMATION	12
3.1	SCA Features.....	12
3.1.1	Push model.....	12
3.1.1.1	Overview.....	12
3.1.1.2	External framework management.....	13
3.1.1.3	Registered and obtainable provides ports	14
3.1.2	Enhanced Application Connectivity	16
3.1.2.1	Background.....	16
3.1.3	Nested applications.....	17
3.1.3.1	Use cases for nested applications.....	17
3.1.3.2	How nested applications work in SCA.....	18
3.1.4	Application Interconnection	20
3.1.4.1	Overview.....	20
3.1.4.2	Use case for interconnecting applications	21
3.1.4.3	Application interconnection design	21
3.1.4.4	Application interconnection implementation.....	21
3.1.4.5	ApplicationFactoryComponent support for interconnected applications	22
3.1.5	Enhanced allocation property support	23
3.1.5.1	Overview.....	23
3.1.5.2	Descriptor structure for nested applications	23
3.1.5.3	SCA Enhanced Allocation Properties.....	24
3.1.5.4	SCA Dependency Hierarchies	25
3.1.6	Lightweight Components.....	28
3.1.6.1	Overview.....	28
3.1.6.2	Benefits	29
3.1.6.3	SCA Solution	29
3.1.6.4	Implementation Considerations	29
3.1.7	Component Model	30
3.1.7.1	Overview.....	30
3.1.7.2	Interfaces and Components.....	31
3.1.7.3	Benefits and Implications	32
3.1.8	Units of Functionality and SCA Profiles	34

3.1.8.1	Overview.....	34
3.1.8.2	SCA UOFs and Profiles.....	34
3.1.8.3	Use of UOFs and Profiles.....	35
3.1.9	Late Registration.....	36
3.1.9.1	Application Registration.....	37
3.1.9.2	PlatformComponent Registration.....	37
3.1.9.3	Late Registration.....	38
3.1.10	Enhanced Process Collocation Support.....	39
3.1.10.1	Background.....	39
3.1.10.2	Earlier SCA Capabilities.....	40
3.1.10.3	Enhanced SCA Capabilities.....	40
3.1.11	Self-Launching Components.....	40
3.2	Design Guidance.....	41
3.2.1	CORBA profiles.....	41
3.2.1.1	Guidance on the use of Any.....	41
3.2.1.2	Guidance on the availability of commercial ORBs implementing these profiles.....	41
3.2.1.3	Use Case for the Lightweight profile.....	41
3.2.1.4	Guidance on restriction interface data types.....	43
3.2.1.5	Rationale for CORBA feature inclusion in the profiles.....	43
3.2.2	SCA Waveform Construction.....	44
3.2.2.1	Overview.....	44
3.2.2.2	FM3TR waveform example.....	44
3.2.3	Static Deployment.....	46
3.2.3.1	Overview.....	46
3.2.3.2	Deployment Background.....	46
3.2.3.3	Connection Management.....	47
3.2.3.4	Example.....	48
3.2.4	Application PIM Profiles Conformance Benefits.....	48
3.2.4.1	Application Conformance.....	48
3.2.4.2	Engineering Tool Conformance.....	49
3.2.5	IDL PSM Constraints.....	49
3.2.6	Organization Specific SCA Tailoring.....	49
3.2.6.1	Organization Specific Interfaces.....	50
3.2.6.2	Organization Specific Components.....	51
3.2.6.3	Organization Specific Components - Alternatives.....	52
3.2.6.4	Summary.....	53
3.2.7	Sample Waveform Architecture and Considerations.....	54
3.3	SCA Modifications.....	58
3.3.1	Resource and Device Interface Refactoring.....	58
3.3.1.1	Overview.....	58
3.3.1.2	Resource Related Modifications.....	58
3.3.1.3	Device Related Modifications.....	60
3.3.1.4	Summary.....	63
3.3.2	Refactored CF Control and Registration Interfaces.....	63
3.3.2.1	Overview.....	63
3.3.2.2	<i>DeviceManager</i> Interface Changes.....	63

3.3.2.3	<i>DomainManager</i> interface changes	66
3.3.2.4	<i>Application</i> Interface Changes	67
3.3.2.5	<i>ApplicationFactory</i> Interface Changes	68
3.3.2.6	Summary	68
3.4	Working in an SCA Environment	69
3.4.1	SCA 4.1 Development Responsibilities	69
3.4.1.1	Overview	69
3.4.1.2	Component Development Alignment	69
3.4.1.3	Component Products	70
3.4.2	SCA Maintenance Process – How To Develop a New PSM?	71
3.4.2.1	Overview	71
3.4.2.2	SCA Change Proposal Process – Submitter Roles and Responsibilities	72
3.4.3	SCA Naming Conventions	72
3.4.3.1	Component Naming Conventions	73
3.4.3.2	Interface Naming Conventions	73
3.5	SCA Q&A	75
3.5.1	What elements of OMG IDL are allowed in the PIM?	75
3.5.1.1	Overview	75
3.5.1.2	PIM Background	75
3.5.1.3	PIM usage for SCA developers	75
3.5.1.4	Future PIM evolution	75
3.5.2	What is the Impact of the SCA Port changes?	76
3.5.2.1	Overview	76
3.5.2.2	Port Revisions	76
3.5.2.3	Interface and Implementation Differences	77
3.5.2.4	Implementation Implications	77
3.5.3	Rationale for DeviceManagerComponent Registration	78
3.5.4	Rationale for Removal of Application Release Requirement	78
3.5.5	Removal of the UML to Language Mappings	79
3.6	Future Enhancements	80
3.6.1	Component Life Cycle	80
3.6.1.1	Overview	80
3.6.1.2	BaseComponent State Model < <i>Requesting Additional Input</i> >	80
4	ACRONYMS	81

TABLE OF FIGURES

Figure 1 Example SCA Powered Radio	9
Figure 2 JTR Set and Waveform Interfaces.....	11
Figure 3 Pull model registration	12
Figure 4 Push model registration	13
Figure 5 External framework management	14
Figure 6 Registered port management	15
Figure 7 Obtainable port management.....	15
Figure 8 Port lifecycles	16
Figure 9 Simple nested application.....	17
Figure 10 Security domain divided application.....	18
Figure 11 Inter-application connections	21
Figure 12 Connectivity specific example	22
Figure 13 Inter-application connections with external ports	23
Figure 14 Dependency Hierarchy	26
Figure 15 Dependency Hierarchy and Sub-Applications	27
Figure 16 Allocation property examples	27
Figure 17 Component Optional Composition.....	28
Figure 18 Component Optional Composition.....	29
Figure 19 Optional Composition Design Approaches.....	30
Figure 20 SCA Component Relationships.....	31
Figure 21 SCA Profiles with OE Units of Functionality	36
Figure 22 Application Component Registration.....	37
Figure 23 Platform Component Registration.....	38
Figure 24 Lightweight Component in Lightweight profile	42
Figure 25 Component distributed across multiple processing elements.....	42
Figure 26 Distributed component with FPGA portion	43
Figure 27 Example FM3TR SCA Waveform Design.....	45
Figure 28 Example Deployment of FM3TR.....	46
Figure 29 ApplicationFactory Role in Component Deployment.....	47
Figure 30 Device Component Definition.....	50
Figure 31 Definition of an Organization Specific Interface	51
Figure 32 Use of an Organization Specific Interface	51
Figure 33 Base Component Definition	52
Figure 34 Model of an Organization Specific Component.....	53
Figure 35 High Level APCO-25 Architecture	54
Figure 36 APCO-25 Platform Components.....	55
Figure 37 <i>Resource</i> Interface Refactoring	58
Figure 38 Application Component Optional Interfaces.....	59
Figure 39 <i>ResourceFactory</i> Interface Refactoring	60
Figure 40 <i>Device</i> Interface Inheritance Refactoring.....	61
Figure 41 <i>Device</i> Interface Refactoring.....	61
Figure 42 <i>LoadableDevice</i> Interface Refactoring.....	62
Figure 43 <i>ExecutableDevice</i> Interface Refactoring	62
Figure 44 <i>DeviceManager</i> Interface Refactoring – registration operations	64

Figure 45 <i>DeviceManager</i> Interface Refactoring – attributes	65
Figure 46 <i>DeviceManager</i> Interface Refactoring – miscellaneous operations	65
Figure 47 <i>DomainManager</i> Interface Refactoring – registration operations	66
Figure 48 <i>DomainManager</i> Interface Refactoring – manager registration operations.....	67
Figure 49 <i>DomainManager</i> Interface Refactoring – installation operations	67
Figure 50 <i>ApplicationManager</i> Interface Refactoring.....	68
Figure 51 <i>ApplicationFactory</i> Interface Refactoring.....	68
Figure 52 General Allocation of Components to Radio Developers	69
Figure 53 SCA Change Proposal Process.....	71
Figure 54 SCA Components	73
Figure 55 SCA Interfaces	74
Figure 56 Port Interface Refactoring	76
Figure 57 Port Implementation Differences	77
Figure 58 Sequence Diagram depicting application release behavior	79
Figure 59 Component Life Cycle	81

1 SCOPE

This User's Guide is intended to provide practical guidance and suggestions for developing Software Communications Architecture (SCA) compliant products. It is not a substitute for the SCA specification, but a companion document to provide implementation guidance and design rationale which complement the formal specification. This document will expand in content and detail as SCA user experiences accumulate.

1.1 INFORMATIVE REFERENCES

The following documents are referenced within this specification or used as reference or guidance material in its development.

- [1] Software Communications Architecture Specification Appendix B: SCA Application Environment Profiles, Version 4.1, 20 August 2015.
- [2] OMG Document formal/2012-11-12, Common Object Request Broker Architecture (CORBA) Specification, Version 3.3 Part 1: CORBA Interfaces, Version 3.3, November 2012.
- [3] OMG Document formal/2008-11-06, Common Object Request Broker Architecture (CORBA) for embedded Specification, Version 1.0, November 2008.
- [4] Software Communications Architecture Specification Appendix E-2 - Attachment 1: SCA CORBA Profiles (from CORBA/e), Version 4.1, 20 August 2015.
- [5] Software Communications Architecture Specification Appendix D - Platform Specific Model (PSM) - Domain Profile Descriptor Files, Version 4.1, 20 August 2015.
- [6] Software Communications Architecture Specification Appendix F - Units of Functionality and Profiles, Version 4.1, 20 August 2015.
- [7] OMG Document formal/2002-04-01, UMLTM Profile for CORBATM Specification, Version 1.0, April 2002.
- [8] Software Communications Architecture Specification Appendix E: Model Driven Support Technologies, Version 4.1, 20 August 2015.
- [9] Donald R. Stephens, Cinly Magsombol, Chalena Jimenez, "Design patterns of the JTRS infrastructure", MILCOM 2007 - IEEE Military Communications Conference, no. 1, October 2007, pp. 835-839.
- [10] Cinly Magsombol, Chalena Jimenez, Donald R. Stephens, "Joint tactical radio system—Application programming interfaces", MILCOM 2007 - IEEE Military Communications Conference, no. 1, October 2007, pp. 855-861.
- [11] Donald R. Stephens, Rich Anderson, Chalena Jimenez, Lane Anderson, "Joint tactical radio system—Waveform porting", MILCOM 2008 - IEEE Military Communications Conference, vol. 27, no. 1, November 2008, pp. 2629-2635.
- [12] JTRS Waveform Portability Guidelines,
<http://www.public.navy.mil/jtnsc/sca/Pages/portabilityguidelines1.aspx>.
- [13] JTRS Open Source Information Repository, http://gforge.calit2.net/gf/project/jtrs_open_ir/.
- [14] Anthony Nwokafor, "Design and implementation of an encryption framework for APCO P25 using an open source SDR platform in an OSSIE environment", Master's Thesis, University of California San Diego, 2012.

2 SCA INTRODUCTION

2.1 SEPARATION OF WAVEFORM AND OPERATING ENVIRONMENT

A fundamental feature of the SCA is the separation of waveforms from the radio’s operating environment. Waveform portability is enhanced by establishing a standardized host environment for waveforms, regardless of other radio characteristics. An example diagram of an SCA-based radio is illustrated within Figure 1. The waveform software is isolated from specific radio hardware or implementations by standardized APIs.

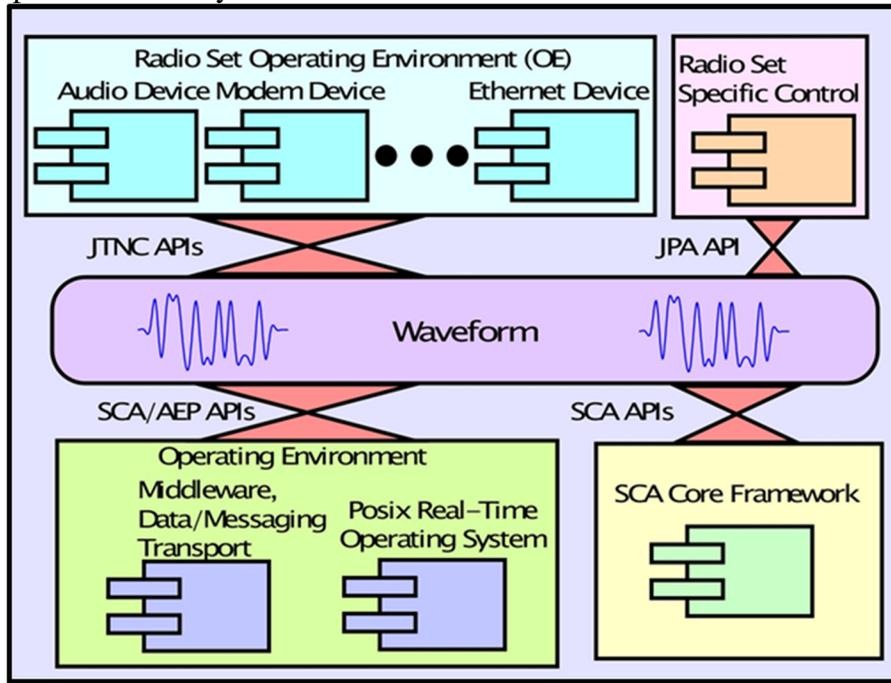


Figure 1 Example SCA Powered Radio

2.2 OPERATING ENVIRONMENT

2.2.1 Application Environment Profiles

To promote waveform portability among the many different choices of operating systems, the SCA specifies the operating system (OS) functionality relative to IEEE POSIX[®] options and units of functionality. The Application Environment Profiles (AEP) specification [1] identifies specific operations such as `pthread_create()`, `open()`, etc., that are available for use by ManageableApplicationComponents and must be provided by the radio platform. A platform may implement or provide additional OS functions, but waveform access to those functions is constrained to those defined in the AEP profiles. This prohibition ensures that any SCA compliant radio can support the waveform’s OS calls.

The SCA AEP defines three profiles, the AEP, Lightweight (LwAEP) and Ultra-Lightweight (ULwAEP) that may be used across a range of radio sets ranging from a small handheld to a multichannel radio embedded within an aircraft. The LwAEP is a subset of the AEP and intended

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

for constrained processors such as Digital Signal Processors (DSP)s that typically do not support more capable real-time operating systems. The ULwAEP is a subset of the LwAEP and intended for very constrained, microkernel based systems.

Some waveforms may require networking functions such as *socket* or *bind*. If a radio platform is going to host waveforms that utilize those operations, it must support the Networking Functionality AEP as an extension to the primary AEP profile. Reference [4] provides additional information related to networking.

2.2.2 Middleware and Data Transfer

In Figure 1, the radio platform provides middleware and data/messaging transport in addition to the real-time operating system. Middleware is a generalized service which facilitates messaging between software components which may or may not be hosted on separate processors. SCA 2.2.2 and its predecessors mandated CORBA as the middleware layer and delegated the choice of a specific transport protocol to the radio set developer. Common data transfer protocols are TCP-IP and shared memory. The former can introduce substantial latency and may have unfairly tarnished CORBA's reputation within the radio community. Faster transports such as shared memory generally yield latencies more acceptable to high-data rate waveforms.

SCA does not have a CORBA requirement and defines middleware independent APIs, although they are still specified in interface definition language (IDL) [2]. Radio developers may continue to use CORBA or select a different middleware such as the lightweight Remote Procedure Call (RPC) used by the Android platform. If an alternate middleware is selected, then products that were dependent on the prior mechanism would require recompilation, but the APIs should remain the same for the most part, thus maximizing waveform portability.

2.3 JTNC APPLICATION PROGRAM INTERFACES

Figure 1 contains several independent APIs which separate the waveform from the radio set. The primary emphasis of the JTNC API standardization efforts has been upon interfaces between the waveform and radio set such as those illustrated in Figure 2. The internal interfaces and transport mechanisms of the radio are defined as necessary by the radio provider. The underlying intent is to provide portability or reuse of the waveform between radio platforms and not necessarily portability of the radio operating environment software. For additional discussion on waveform portability, see [11] and [12].

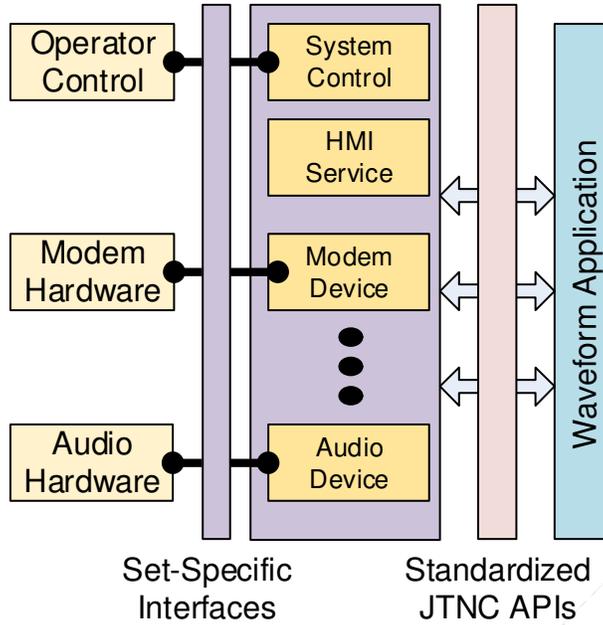


Figure 2 JTR Set and Waveform Interfaces

There has been a conscious effort to maintain a clear separation between the SCA and the JTNC APIs which define services provided by the radio set to the waveform such as GPS, time, etc. The distinction not only maintains the integrity of SCA framework and preserves its applicability across a wide range of domains, but also allows the content of each family of specifications to evolve according to its own timetable. A partial list of the JTNC APIs is provided in Table 1. The APIs have been developed with software design patterns that encourage a scalable and extensible infrastructure. See [9] and [10] for an introduction to the aggregation, least privilege, extension, explicit enumeration, and deprecation design patterns used by the JTNC APIs.

Table 1 Partial List of JTNC APIs

Audio Port Device API	Ethernet Device API
Frequency Reference Device API	GPS Device API
Modem Hardware Abstraction Layer (MHAL) API	Serial Port Device API
Timing Service API	Vocoder Service API
MHAL On Chip Bus (MOCB) API	Packet API
JTRS Platform Adapter (JPA) API	

The JTRS Platform Adapter (JPA) is both an API and a design pattern for controlling the waveform by the radio set (it is a particularly vexing problem, to define a portable command/control interface for waveforms across multiple radio sets). This API uses the SCA *PropertySet* interface as a container for waveform parameters controlled and manipulated by the radio set. It also supports bidirectional communication, permitting the waveform to provide status to the radio set.

3 TOPIC ORIENTED GUIDANCE AND SUPPLEMENTARY INFORMATION

3.1 SCA FEATURES

3.1.1 Push model

3.1.1.1 Overview

Earlier SCA versions were pull model oriented as shown in Figure 3. References are exchanged between providers and consumers, but callbacks are required to retrieve information from the provider component.

For example:

- *getPort* for pulling uses and provides ports
- Pulling attributes (e.g. deviceID, registeredDevices)
- Pulling Application Components from a Naming Service

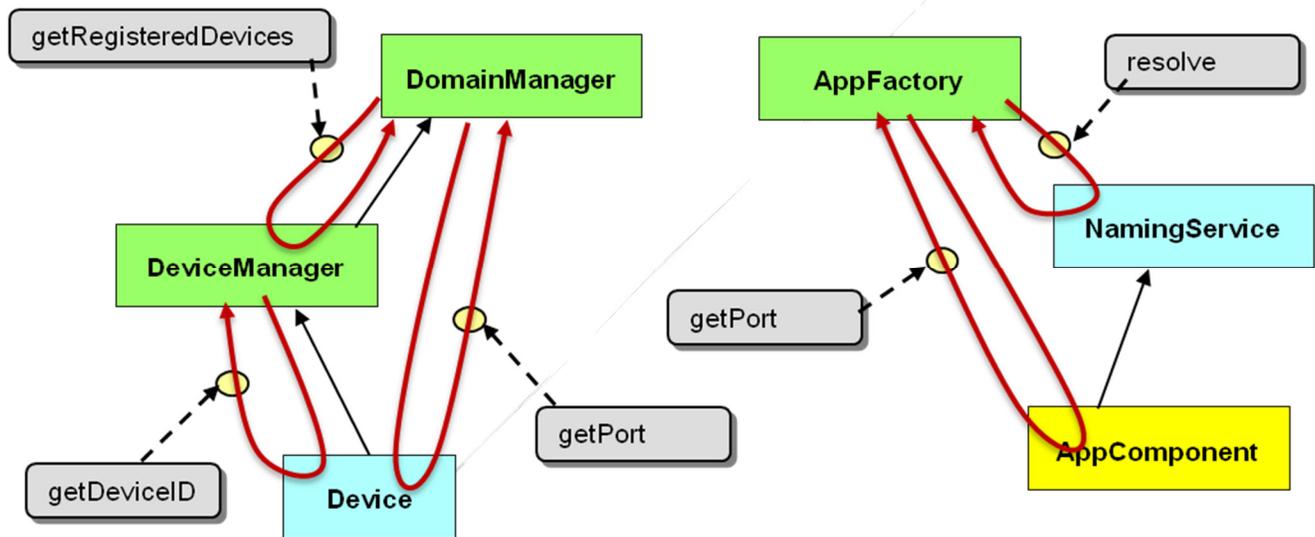


Figure 3 Pull model registration

SCA now utilizes a push model, Figure 4, architectural approach that allows for a direct exchange of information without callbacks. The primary benefits of this model are better information assurance and performance. Better information assurance is achieved by limiting component to manager access to pushes only and eliminating the need for a Naming Service. Performance is enhanced as the total number of calls involved in the registration process is reduced. This can decrease component startup and instantiation time. Push model registration also allows the call back attributes and operations to become optional and when they are not used the amount of required implementation can be reduced.

For example:

- Device ID and Provides Ports can be pushed with the data provided at component registration time and don't need to be pulled later
- Registered components (complete with IDs and Provides Ports) can be pushed with DeviceManagerComponent registration

- The DCD information can also be pushed instead of pulled by accessing a DeviceManagerComponent attribute
- Direct registration of application components removes the need for a Naming Service

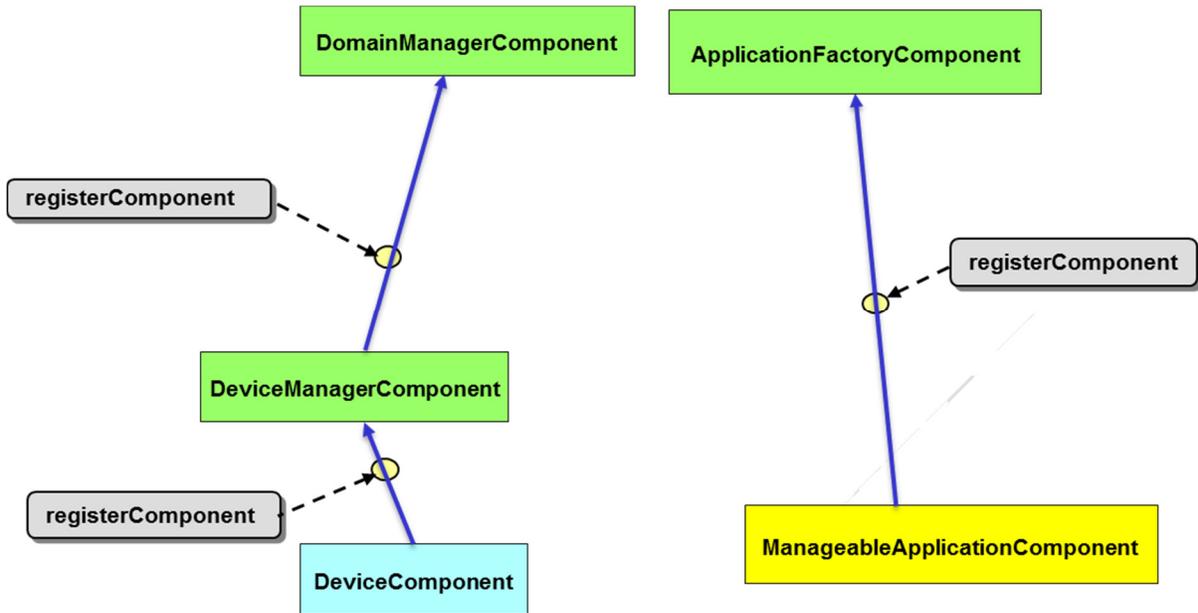


Figure 4 Push model registration

3.1.1.2 External framework management

External Framework Management was expanded slightly to accommodate a push model. For example

- The *installApplication* return now provides a ComponentType data structure that contains data elements which previously required separate pull calls.

However, external framework management predominately maintains the pull model support of previous SCA versions.

The rationale for this approach is that it provides a good balance of performance, capability and compatibility. It affords greater performance when utilizing the push model extension for external management, but continues to support the existing use cases where pulls may still be needed. It also allows for backward compatibility without violating the least privilege principle.

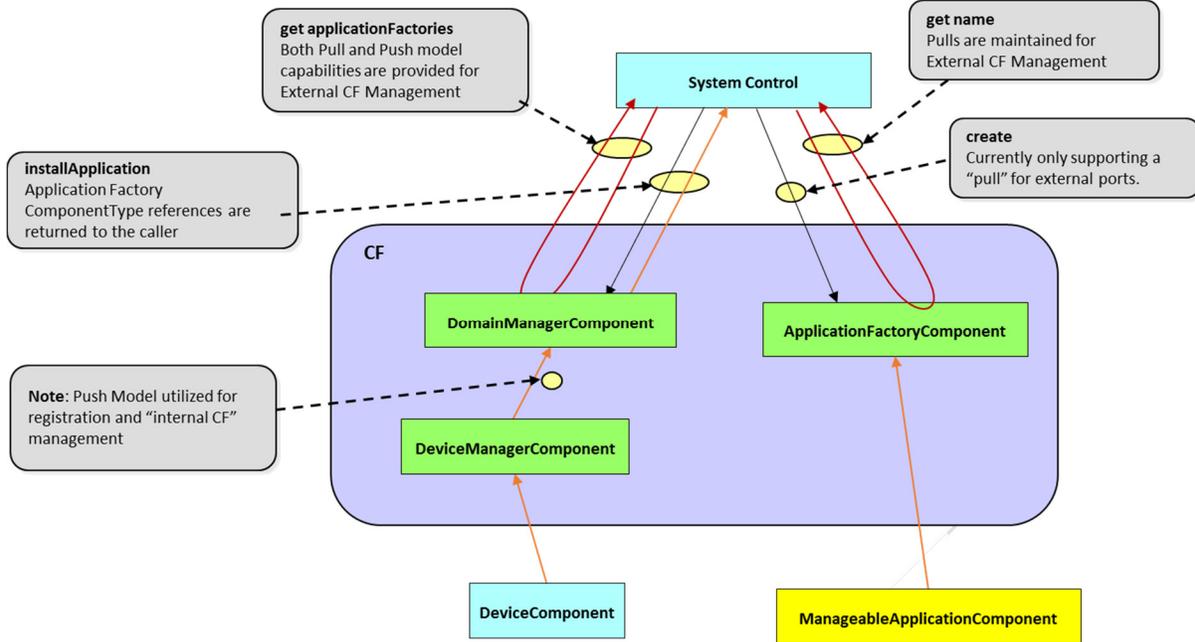


Figure 5 External framework management

3.1.1.3 Registered and obtainable provides ports

In order to implement a push model and allow continued support of prior use cases, the provides port semantics had to be enriched. SCA currently provides two types of provides ports, “Registered” and “Obtainable”. Sometime these are referred to using the terms “Static” and “Dynamic” which are found in earlier SCA versions. To avoid confusion, Registered Provides ports = Static Provides Ports. Obtainable Provides Ports = Dynamic Provides Ports.

3.1.1.3.1 Registered provides ports

Registered provides ports are provides ports which have a lifecycle tied to the lifecycle of the component. Registered ports are registered with the framework during component registration and the framework will not attempt to retrieve them when making connections. Registered ports are not explicitly released by the framework except through the component’s *releaseObject* operation. Thus, the *getProvidesPorts* and *disconnectPorts* operations typically are not called for registered provides ports. For assurance reasons, there may be cases when a component may want to reject calls for these ports explicitly (e.g. raise an *UnknownPort* or *InvalidPort* exception). There may also be instances when a component may want to allow ports that are “registered” to still also be “obtainable”. Meaning the ports can be retrieved from *getProvidesPorts* and then connections to the ports can be disconnected through *disconnectPorts*. The exact details surrounding the semantics of port connectivity are left unspecified to allow component developers to customize this behavior to match the needs of the target platform.

However a framework that is built in accordance with the specified SCA requirements will not retrieve registered provides ports through *getProvidesPorts* and will not disconnect them through *disconnectPorts*.

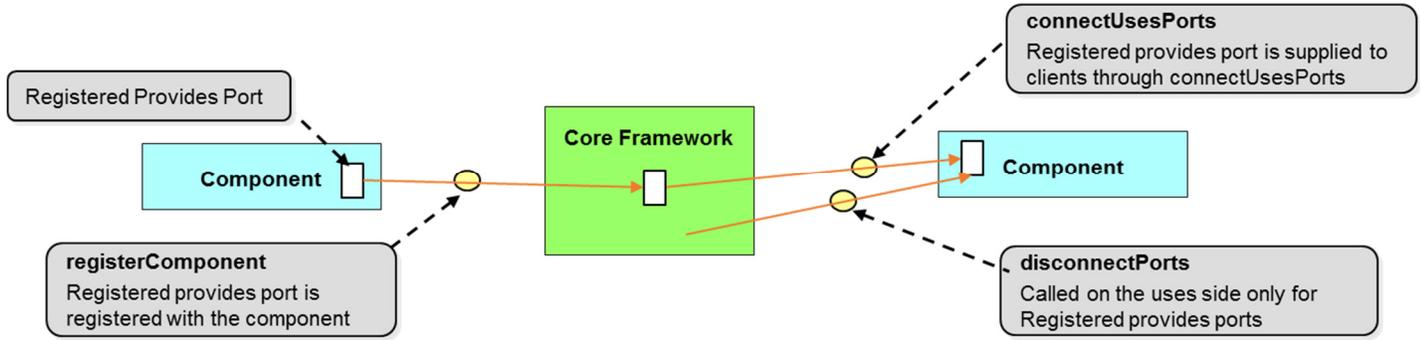


Figure 6 Registered port management

3.1.1.3.2 Obtainable provides ports

Registered provides ports are provides ports which are meant to have a lifecycle tied to the lifecycle of a given connection. Obtainable provides ports are not registered with the component and instead the framework will attempt to retrieve the ports through the *getProvidesPorts* operation when they are needed to complete connections. Obtainable provides ports are explicitly released by the Framework via the *disconnectPorts* operation when the connections to them are torn down. With obtainable provides ports, by specifying connectionIDs on *getProvidesPorts* and calling *disconnectPorts*, additional use cases and added functionality are supported that is not available within prior SCA versions.

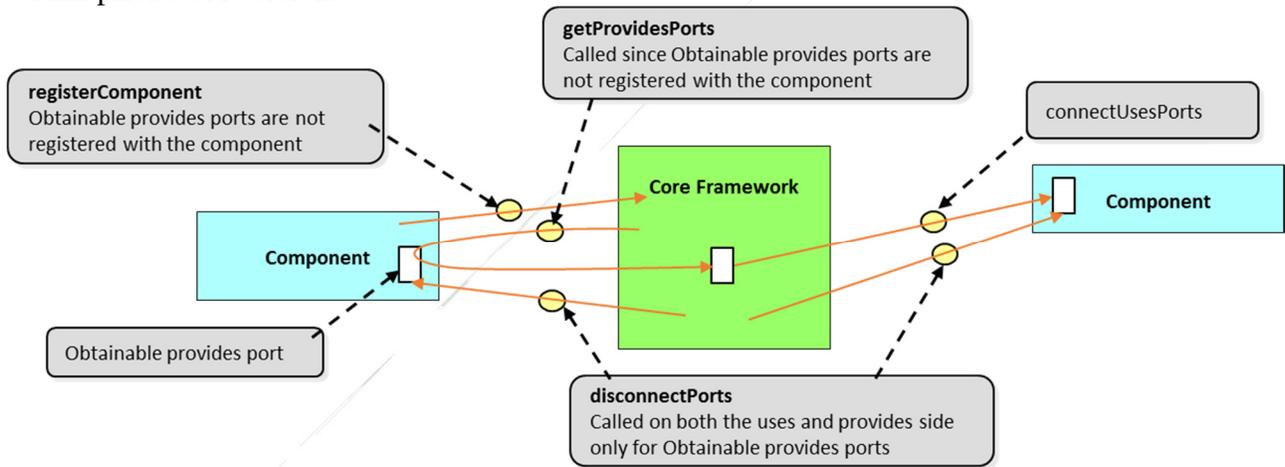


Figure 7 Obtainable port management

Whether or not obtainable provides ports have to be tied to the lifecycle of a given connection is not specified. Several use cases exist where they may have a longer lifecycle:

- A “backward compatibility” use case where a provides port is created and released with the component, but not registered, mimicking the prior SCA pull-model behavior
- A “fan in” use case where the same provides port instance services multiple connections, with reference counting used to dictate when it is released.

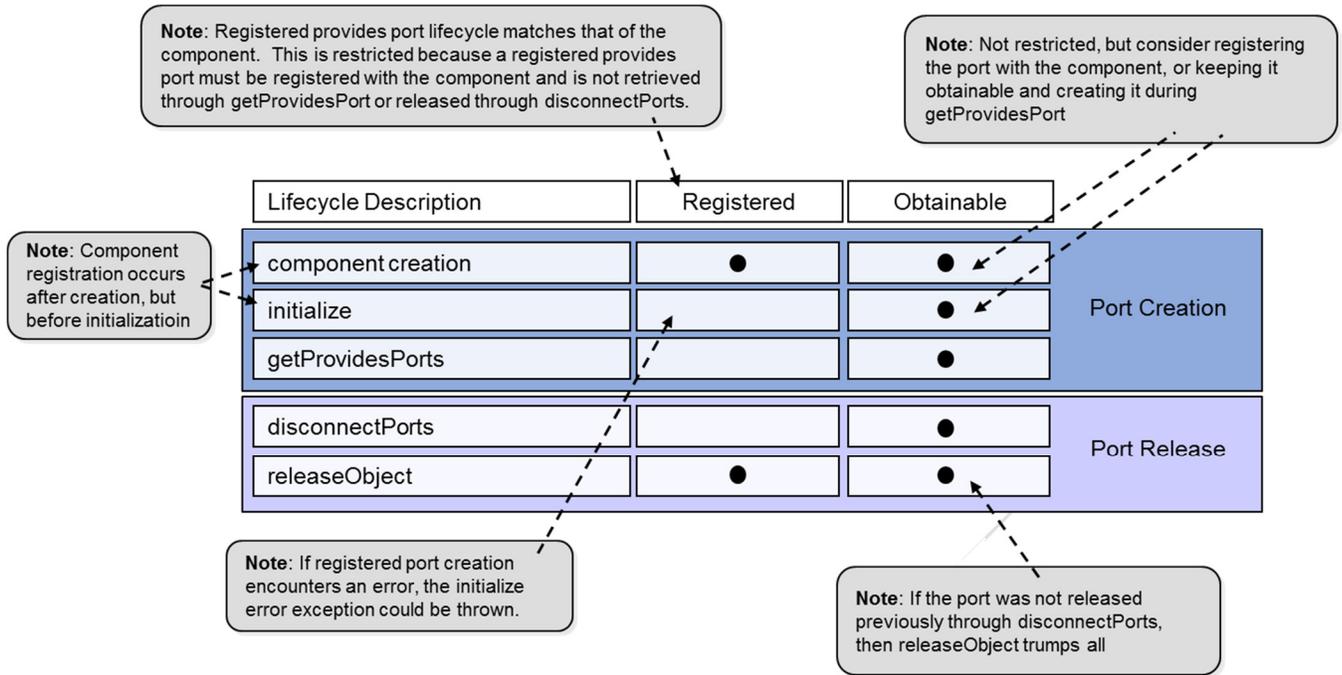


Figure 8 Port lifecycles

3.1.2 Enhanced Application Connectivity

3.1.2.1 Background

Prior SCA releases only supported the ability to deploy individual, standalone applications. While multiple applications could be deployed on a platform, the SCA component framework did not provide direct support to interconnect or logically nest those applications. As a result, the client creating the applications was left to do this manually, using a combination of external ports and either “hard coded” interconnection or automatic interconnection using information gleaned from the application XML.

However, this approach was very limited and required much of the client. Since endpoint interconnection was not automatically controlled by the SCA a number of challenges existed, such as the following:

- Added complexity to client code – the client code needs to understand how to retrieve and establish port connections, and for some implementations utilize XML to introspect the application information.
- Reduced security – in some systems, the ability to make CORBA port connections is intentionally restricted to preserve application integrity, and for similar reasons, the ability to obtain the necessary CORBA object references is restricted.
- Abstraction / Information hiding – in some cases, you may want an application to behave like a single component, and include such a sub-application within an outer component. Pre-SCA 4 frameworks did not support this manner of abstraction
- Distribution of applications – in some systems (typically those with an application partitioned across two or more security domains) it is desirable to decompose an application into sub-applications; with component instantiation and interconnection occurring locally

within the domain, thus minimizing “bypass” traffic crossing domains during creation. In prior versions of the SCA this was not supported, leading to non-optimal workarounds. In the current SCA, a set of capabilities has been added to support the above needs. The two capabilities, “Nested application support” and “Application interconnection” are addressed in the following sections. Nested applications may also benefit from the use of the Enhanced allocation property support, which is described in section 3.1.5.

3.1.3 Nested applications

3.1.3.1 Use cases for nested applications

A simple, monolithic application is still the best solution for many platforms, however several common situations exist where a hierarchical, nested application presents a better solution. The first scenario arises from the simple desire to better manage application instantiation and encapsulate complex application structure into a hierarchical organization. In SCA 2.2.2 and earlier versions the application structure was “flat”, simply consisting of “leaf” components. This limitation no longer exists because complex subassemblies now can be formed and abstracted into sub-applications, which may in turn be combined to form a single application. This architectural technique can enable a subassembly to be used in different contexts, promoting reuse in common asset libraries such as those employed in software product lines.

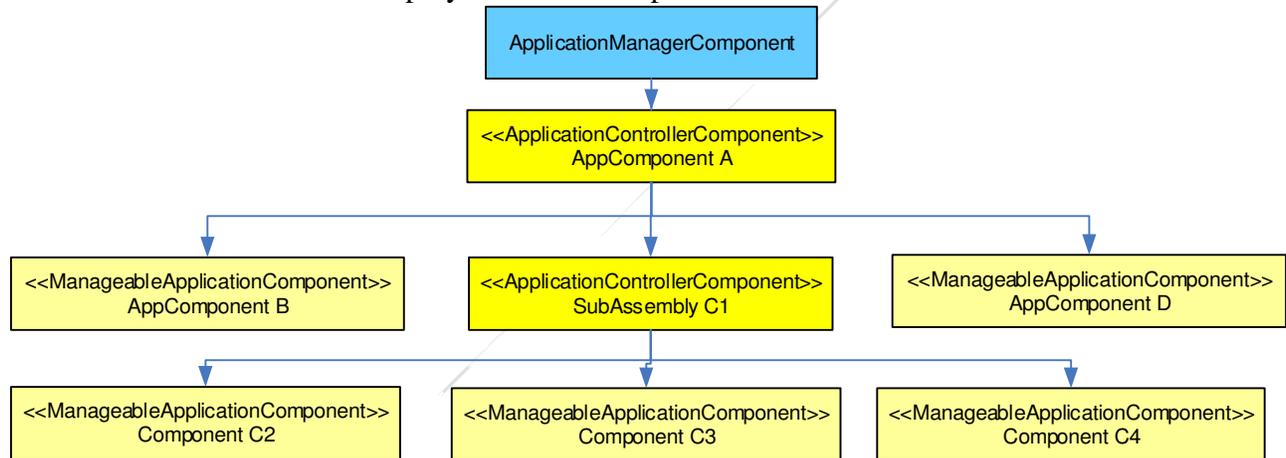


Figure 9 Simple nested application

An example of this composition is shown in Figure 9. In this example, an overall application is made up of four top-level components, with one of the components (AppComponent A) functioning as the application’s ApplicationControllerComponent. Component C1 however is not a simple component created by the normal *componentinstantiation* element within the SAD¹, but rather a sub-application created through an *assemblyinstantiation*. To AppComponentA this nested sub-application is abstracted as a single ManageableApplicationComponent, but from a creational standpoint the “upper level” ApplicationFactoryComponent constructs a true sub-application per a cited SAD. As is discussed later, in this example there is no separate

¹ Componentplacements are located inside either a componentplacement or hostcollocation element

ApplicationManagerComponent produced to manage the sub-application, rather all management is performed by the upper blue ApplicationManagerComponent. However, this approach is a core framework implementation decision, and an equally valid approach would have the sub-application managed by an intermediate ApplicationManagerComponent, through the narrowed interfaces made available by the ManageableApplicationComponent.

A second use-case arises on platforms which provide encryption in such a way that two or more security domains are established (e.g. plaintext and ciphertext domains). In some high assurance environments, these domains are distinct and separated (usually by some sort of cryptographic subsystem) such that control and configuration communication between the domains needs to be minimized. In such a system, it could be beneficial to structure an application such that it resembles two or more independent sub-applications, one in each security domain. A typical representation of this situation is shown in Figure 10.

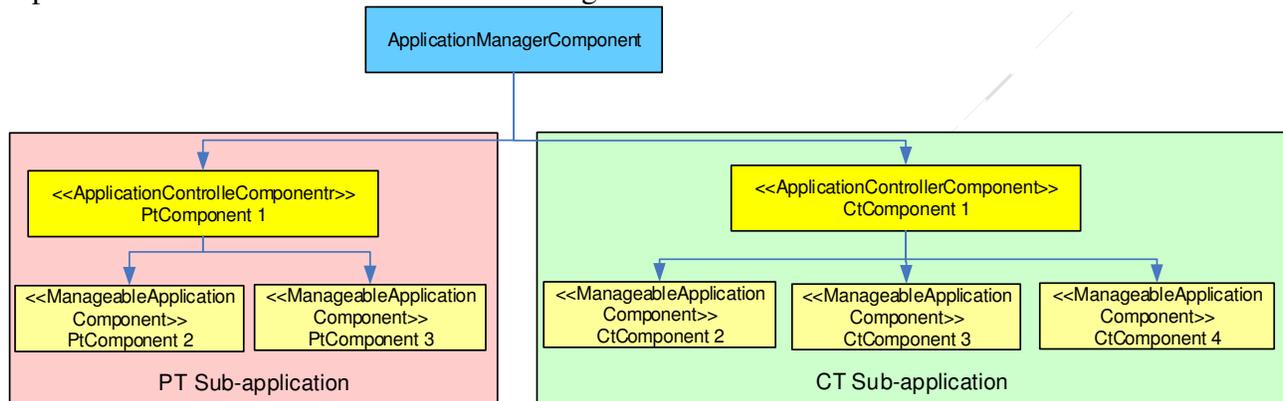


Figure 10 Security domain divided application

In this example, we see a top-level application wholly consisting of two sub-applications, each deployed in a different security domain². The example also has the ApplicationManagerComponent³ distributing properties and controlling two distinct ApplicationControllerComponents. Be aware that the SCA does not specify *how* this application is physically constructed – a clever implementation could distribute the required *CF::ApplicationFactory* behavior across the security domains (while still controlling this through a common *CF::ApplicationFactory* interface) thus minimizing cross-domain communications.

3.1.3.2 How nested applications work in SCA

While a significant enhancement, SCA support of nested applications is not immediately obvious, or described in a dedicated section. Support is enabled through a number of small changes scattered throughout the document. The major modifications required to support this feature exist in Section 3.1.3.3.1.1 (*ApplicationManager*), 3.1.3.3.1.3 (*ApplicationFactory*), and throughout Appendix D.

² Not to be confused with an SCA domain – in this system, there is still only one domain manager.

³ Application ManagerComponents implement the *CF::ApplicationManager* interface and responsibilities, and are created / supplied by the core framework.

3.1.3.2.1 ApplicationFactoryComponent support for nested applications

The ApplicationFactoryComponent, via the *ApplicationFactory* interface, provides the means to create a single, top-level application. The application is created according to the specifications provided in a set of XML files, encapsulated by a Software Assembly Descriptor (SAD), which define how an application will be created. The instructions include which elements are used, and how they are deployed, configured, and connected.

Earlier SCA versions referred to elements as individual components, which were defined by Software Package Descriptors (SPD) and so on. The current SCA adds support for nested applications by allowing not only the creation of components (which could be both “leaf” components and BaseFactoryComponents) but also the creation of assemblies. These assemblies, which function as sub-applications, are represented in the outer SAD by an *assemblyinstantiation* element, itself contained within an *assemblyplacement* element. While the method and order of events is left largely to the implementation, the post-condition is clear – after an application is constructed, all components represented by the outer SAD and those of any child SAD files cited in *assemblyplacements* will have been instantiated, interconnected, and a ComponentType (i.e. ApplicationManagerComponent) returned to the client. Furthermore, only top-level instantiated applications will be listed in the DomainManagerComponent's applications attribute; the presence of any subassemblies is unlisted.

Just as important is what is *not* specified in SCA. Though not an inclusive list, the following implementation alternatives were intentionally preserved:

- SCA does not specify the order in which components and subassemblies should be constructed or initialized.
- SCA neither requires nor prohibits usage of intermediate ApplicationManagerComponents to manage any sub-assemblies. Put another way, in some core frameworks, an implementer could choose to have the top level ApplicationManagerComponent only manage the top level leaf components and delegate any direct subassembly management to a “sub” ApplicationManagerComponent, while in others, a single ApplicationManagerComponent could be responsible for all components.
- SCA does not specify details regarding how nested applications are installed in a system. The DomainManagerComponent's *installApplication()* operation only lists a top level SAD – the deployment of any other necessary files is assumed to have been previously accomplished, and no assumptions are made regarding absolute or relative directory placement.
- The nested SAD is no different from an outer SAD. In this way, an implementation could allow separate installation of the SAD for standalone (“top level”) instantiation, while still allowing the application to be used as a sub-application by citing it from another SAD.
- SCA, while requiring a single client interface (*CF::ApplicationFactory*) and compliance to the requirements of an ApplicationFactoryComponent, does not dictate how the functionality of this component is distributed across the system. In many systems an ApplicationFactoryComponent will map to a single component which singlehandedly guides the deployment. However, other compliant implementations are possible, especially when an application is deployed across processors or security domains. One such example would be a central coordinator which implements the *CF::ApplicationFactory* interface, but delegates some of its component creation behavior to subcomponents (which need not implement any specific interface). This federated deployment could minimize cross processor or cross domain communications in some cases, speeding up deployment, etc.

3.1.3.2.2 ApplicationManagerComponent support for nested applications

The ApplicationManagerComponent⁴ has two broad responsibilities, which were expanded with the introduction of nested applications. The first responsibility is to tear down the application instance created by the corresponding ApplicationFactoryComponent. When nested applications are supported in SCA, the allocation of the teardown responsibilities is unspecified. One implementation approach would be for the top level ApplicationManagerComponent to manage top level components exclusively, with one of them being an ApplicationManagerComponent which manages its sub-application components. The advantage of this approach is one of symmetry (each SAD creates an application and is managed by an ApplicationManagerComponent) and it is most similar to prior SCA core framework implementations. However, other implementations are valid. For example, SCA does not require ApplicationManagerComponents to manage the sub-application components – instead a single, top-level ApplicationManagerComponent could be responsible for tearing down all components (and port disconnection, etc.). This approach may be more efficient in some cases or better centralize the domain data.

Secondly, ApplicationManagerComponents are responsible for distributing client calls made to the Base Application interfaces, specialized by the *CF::ApplicationManager* interface, to the application. In earlier SCA versions distribution was straightforward, all calls were to be passed to a single component which realized the *CF::Resource* interface (not an assembly) that was designated as the *assemblycontroller* in the SAD. If the DMD cardinality attribute has a value of “single”, only one designated *assemblycontroller* exists, and the ApplicationManagerComponent responsibilities remain the same. However in implementations that implement the NestedDeployment UOF and have a DMD cardinality attribute with a value of “multiple”, multiple *assemblycontrollers* are allowed and those *assemblycontrollers* are allowed to refer to an *assemblyinstantiation*. When this is the case, the ApplicationManagerComponent is not able to forward *configure()*, *query()* and *runTest()* as it did before. Instead, it must examine each individual property and forward it to the appropriate *assemblycontrollers* based on the information contained in its top level SAD and derived XML files (which in the nested case would include at least one additional SAD). Additionally, as multiple properties can be listed in a *configure* or *query* call, the ApplicationManagerComponent may also be required to break up those calls, or potentially combine their results and exception behavior.

3.1.4 Application Interconnection

3.1.4.1 Overview

An alternative to having a single, monolithic application would be to have multiple independent applications that collaborate with one another. The SCA application interconnection capability provides a standardized approach for how to address the problem of establishing connections between framework components modeled as applications. Prior to the introduction of this capability multiple solutions were used to address this problem, complicating software reuse and portability. Its introduction should alleviate those problems and ensure that a uniform realization of this approach is available across platforms.

⁴ Prior to the introduction of the SCA Component Model, there was no formal ApplicationManagerComponent, instead all requirements were allocated to an unnamed CF component which implemented in the *CF::ApplicationManager* interface.

3.1.4.2 Use case for interconnecting applications

A scenario which highlights the need for multiple independent applications would be one that requires a system with a clear separation of concerns and loose coupling of components. For example, a radio platform that contains an Android presentation layer which provides a general purpose user interface that manages and monitors the system. This system could have been designed in accordance with the Model, View, Presenter pattern where the applications to be connected would be the waveform (Model) and UI intermediary (Presenter).

Earlier SCA versions did not have a means for the framework to form these connections. The SAD contained the *externalports* element, which by definition provided a means for an application to be connected with components (application or otherwise) external to a waveform, but no corresponding framework guidance or requirements to establish those connections. Typically, the gap was filled by introducing an additional component within the system to perform that functionality.

3.1.4.3 Application interconnection design

The current SCA defines a formal mechanism that utilizes the *externalports* element as the conduit to manage the formation and destruction of those inter-application connections. The external port connection construct provides a good solution because it aligns with the nature of the problem – two applications that need to be connected with one another but they are created independently and there are no guarantees that they will be created. Consequently, the connection mechanism must accommodate instances when one side of the connection does not exist.

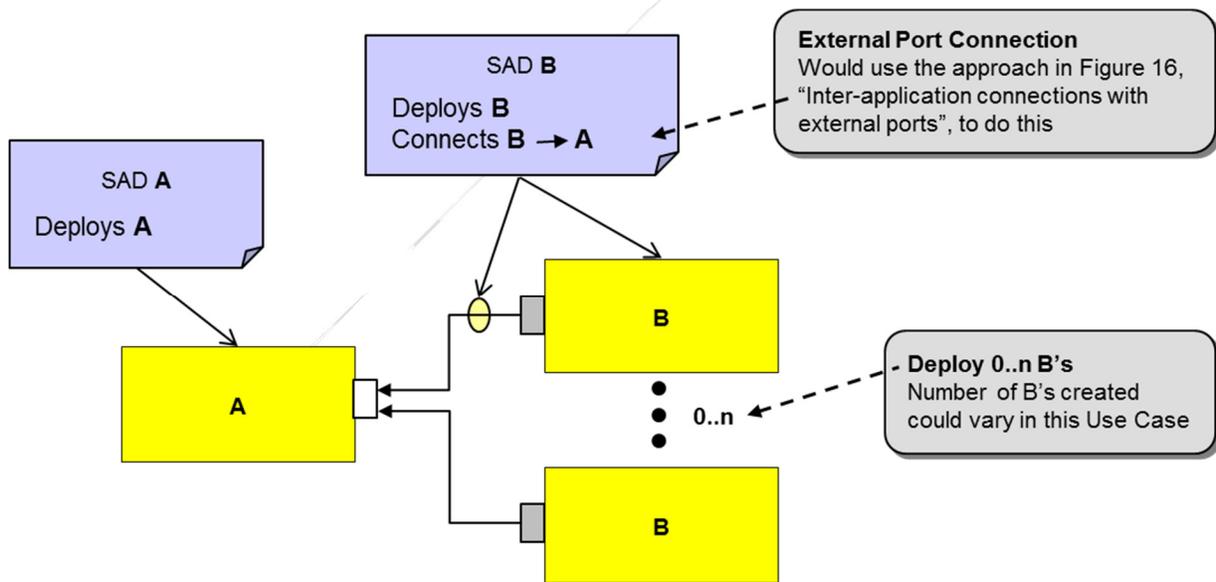


Figure 11 Inter-application connections

3.1.4.4 Application interconnection implementation

Building upon the earlier scenario, both the waveform and the presentation layer will have their connections described in their respective SAD files. The Android presentation layer, application A, contains a provides port that can be accessed and used by other applications, so it will identify that

port within its *externalports* element as a *providesidentifier*. The waveform, application B, wishes to be connected to the presentation layer's external port, so in one of its SAD connections it specifies a connection between its local uses port and the externally provided provides port from A. The example illustrates that only one application needs to define the connection for it to be processed by the framework.

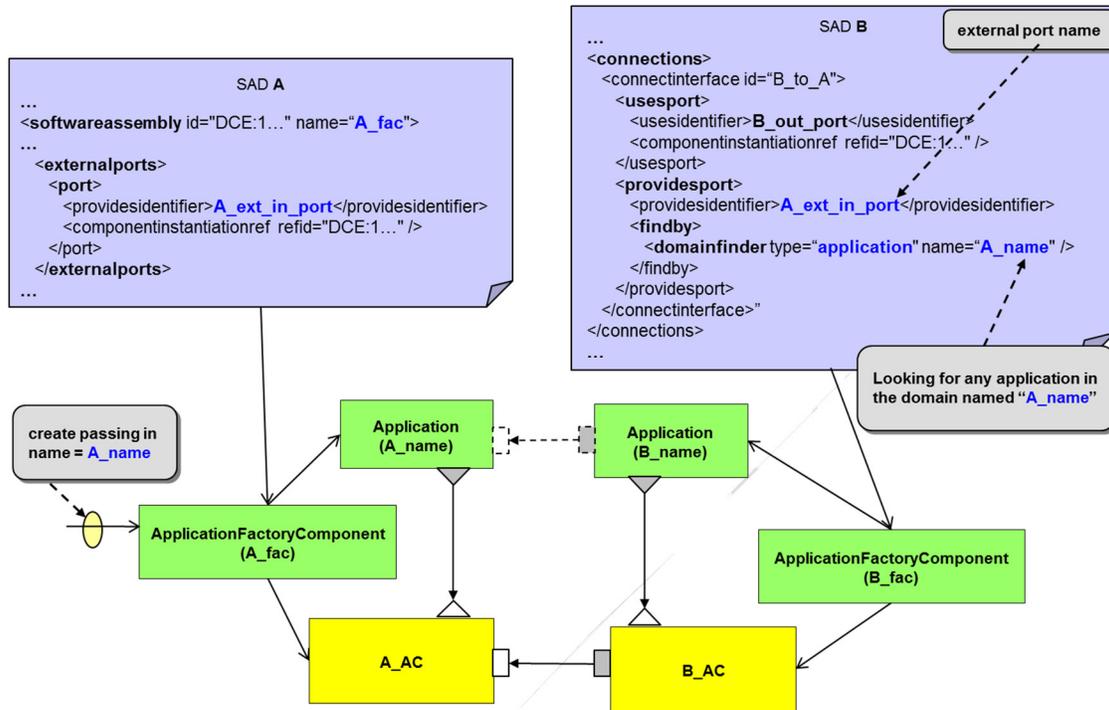


Figure 12 Connectivity specific example

3.1.4.5 ApplicationFactoryComponent support for interconnected applications

SCA now includes an additional type, application, within the *domainfinder* element. The semantics associated with this type provide the framework with information describing the elements that will be included within connections and how those connections should be formed. The ApplicationFactoryComponent retrieves the connection endpoint via the domain's *domainfinder* element. When the application type is used, no implicit creation behavior is intended, so the framework is not expected to instantiate an application if it does not exist. If neither endpoint exists or can be resolved, then the specification permits implementation specific behavior. However, the desired approach in the aforementioned scenario would be for the connection to be held in a pending state until it can be established (note that in this approach either the waveform or the framework will need to have sufficient safeguards to ensure that a call to this connection prior to its formation does not result in an unexpected or uncontrolled termination). An alternative course of action would be to prevent the application from being instantiated, although this seems excessive as a well-designed waveform should not have critical dependencies that exist across application boundaries.

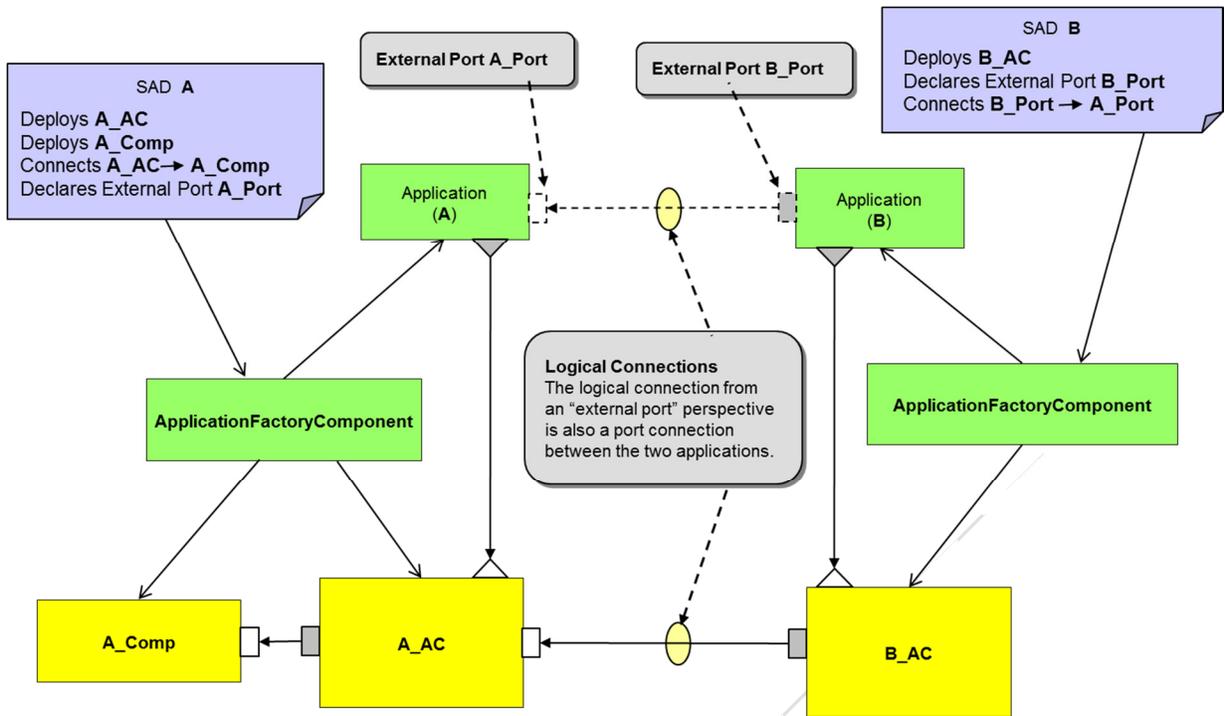


Figure 13 Inter-application connections with external ports

The ApplicationFactoryComponent must be able to accommodate multiple connection strategies depending on the information provided in the domain profile. When only the application name is specified, any ApplicationManagerComponent in the domain with that name can be used. When both the application factory and application names are specified, only the named ApplicationManagerComponent created by the specified ApplicationFactoryComponent may be used. When only the application factory name is specified then any ApplicationManagerComponent created by the specified ApplicationFactoryComponent may be used.

3.1.5 Enhanced allocation property support

3.1.5.1 Overview

Several use cases exist that require the framework to have the ability to constrain the deployment of application or nested application components. SCA 2.2.2 provided this capability with the channel deployment functionality contained within the Software Communications Architecture Extensions specification. Those capabilities were included within this SCA revision, and an alternative approach was provided with the introduction of nested applications. Nested applications extend SCA 2.2.2 allocation properties by making them more dynamic and accessible to nested applications. The new constructs provide users with the ability to deploy nested applications to different domains.

3.1.5.2 Descriptor structure for nested applications

The SAD's definition was modified in this SCA release to accommodate nested applications. An SCA application consists of 0 or more components and 0 or more nested applications. The nested

applications incorporate a new element, *applicationinstantiation*, which is similar to a *componentinstantiation*, but has different sub-elements.

Nested applications are similar to a *ManageableApplicationComponent* in that they can receive *properties*, *deviceassignments* and *deploymentdependencies*. However they differ from those components in that they cannot be created by a *BaseFactoryComponent*. The information in the *applicationinstantiation* element is intentionally similar to the *ApplicationFactory::create()* call. This similarity permits an implementation to use the *ApplicationFactory::create()* operation to create a nested application.

```

<!ATTLIST componentfile
  id ID #REQUIRED
  type CDATA #IMPLIED>
<!ELEMENT partitioning
  ( componentplacement | hostcollocation
    | assemblyplacement )+>
<!ELEMENT assemblyplacement
  ( componentfileref
    , assemblyinstantiation+ )>
<!ELEMENT assemblyinstantiation
  ( componentproperties? ,
    , deviceassignments? ,
    , deploymentdependencies? ,
    , executionaffinityassignments? ) >
<!ATTLIST assemblyinstantiation
  id ID #REQUIRED>

```

Type can be "software package descriptor" or "software assembly descriptor"

Assemblies may consist of both components and assemblies (e.g. SAD). However, assemblies cannot be inside *hostcollocation* sections and cannot be created by component factories.

New element, modeled after *componentinstantiation*. *Componentproperties* (configureproperty type only), override nested SAD similar to that in create call and *deviceassignments* and *deploymentdependencies* act in the same way as if passed into *ApplicationFactory::create()*.

Nested assemblies can also serve as application controllers

3.1.5.3 SCA Enhanced Allocation Properties

SCA 2.2.2 allocation properties could only be assigned in .prf files, and not overridden. Similarly, dependencies were specified in .spd files, and could not be overridden. This severely limited the manner in which they could be used.

The SCA deploys components by evaluating dependency requirements against existing component allocation property definition. As an example a *DeviceComponent* (or other component) defines an allocation property in a .prf file as follows:

```

<simple id="RadioChannel" type="short" name="RadioChannel">
  <value>0</value>
  <kind kindtype="allocation"/>
  <action type="eq"/>
</simple>

```

Then a component to be deployed establishes a dependency against the allocation property by stating the type of device it requires:

```

<dependency type="RadioChannelDependency">
  <propertyref refid="RadioChannel" value="5"/>
</dependency>

```

If the dependency can be satisfied by one of the component allocation property definitions within the domain, then that DeviceComponent becomes a usage or deployment candidate.

SCA now provides the ability to override component allocation properties in the *componentinstantiation* section. This allows a system designer to assign different values to allocation properties on a per-instance basis, e.g. “the channel 4 instance of the GppDevice gets the deployedChannel allocation property overridden to 4”. In prior SCA versions, a system designer would have had to edit the component's .prf file or use the SCA extension .pdd file to accomplish this. SCA also introduced a capability to specify SAD and *create()* based *deploymentdependencies*. The *deploymentdependencies* element specifies a list of dependencies which can override SPD defined dependencies (either within deployment or as part of a uses device connection). The dependency relationship is overridden, not the allocation property, which differs from other “property overrides”. Lastly, a list of *deploymentdependencies* can be passed into the *ApplicationFactory::create()* operation to allow client-controlled dependencies (e.g. radio channel) to be specified.

3.1.5.4 SCA Dependency Hierarchies

SPDs define the dependencies for a particular component type. Unless overridden, these definitions apply to all instances of the component.

As shown in Figure 14, SAD *componentinstantiations* can optionally override a dependency for a given instance – if the SPD uses the dependency for deployment or a *usesdevice* relationship. This would, for example, allow an application to place two instances of the same component in different domains.

An optional top-level SAD *deploymentdependencies* element allows for global dependency overriding across all applicable application components (see Figure 14). Using this approach does not impose the dependency on a component, but overrides it as if a like-named dependency existed within the component's SPD. This approach is likely more applicable within an assembly that uses nested applications.

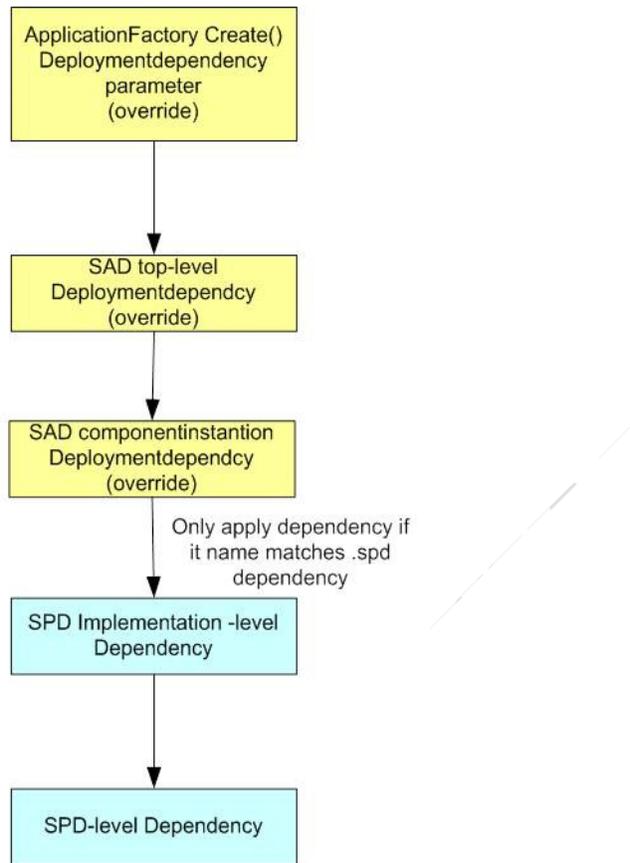


Figure 14 Dependency Hierarchy

At the highest level of the dependency hierarchy, a client could supply *deploymentdependencies* which could be applied to the entire application. A common usage scenario would be to specify a radio channel placement dependency. As Figure 15 depicts, when application nesting is used, the rules stay the same but overriding occurs from the outermost SAD (highest precedence) to the innermost SAD. An additional *deploymentdependency* is added to the *assemblyinstantiation* element. This allows dependencies to be supplied that would apply to the nested application (and any of its children). A common usage scenario for this capability would be to place distinct sub-applications in different domains.

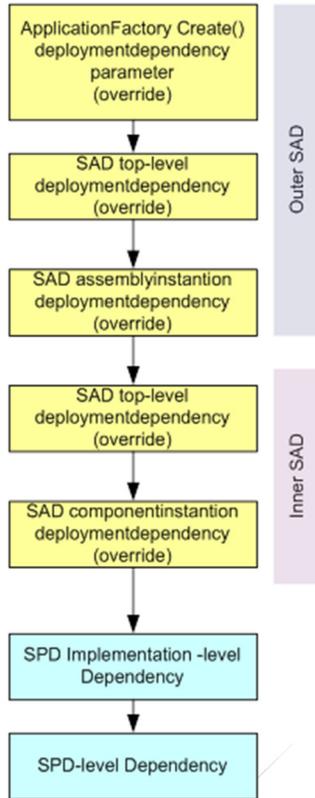


Figure 15 Dependency Hierarchy and Sub-Applications

The following table provides an example of a class of allocation properties and how they might be used within a system:

Element	Typical use	Example
ApplicationFactory Create() by client	Controls placement of an specific application instance. Typical use would be placement on a specific radio channel and/or domain	radioChannel eq 2
Outer .sad top-level	Uncommonly used. Controls placement of an overall application that is not instance specific	
Outer .sad assemblyinstantiation	Controls placement of a given nested application instance.	domain eq "green" (if instance specific)
Nested .sad top-level	Controls "hard coded" placement of a nested application. Used when instance-specific overriding is not used / needed. Typical use would be for forcing location to a specific domain	domain eq "purple" (if fixed for application)
Nested .sad componentinstantiation	Uncommonly used	
.spd dependency element	Defines dependencies actually needed by a component. Note that if not specified, cannot be overridden. "Default" values allowed	radioChannel eq 0 domain eq "white"
.prf allocationproperty DEFINITIONS	Set in .spd / .scd prf files, can be overridden at component instantiation	radioChannel = 5 domain = "blue"

Figure 16 Allocation property examples

3.1.6.2 Benefits

Each optional composition flag shown in the Unified Modeling Language (UML) is associated with a UOF in Appendix F [6]. Having the ability to eliminate unnecessary interfaces allows components to be smaller and more focused than components realized in accordance with earlier SCA versions. Having fewer interfaces to realize reduces a component's footprint size; one should remember that there are size implications associated with stubbed implementations. The savings realized from a single component might be minimal, but the amount can add up when extended across all of the components that comprise a radio set. Omitting rather than stubbing unneeded operations can also improve a system's assurance profile because it eliminates a potential vulnerability of having an additional system operation, in this case one that might be given less scrutiny because it was not intended to be used. Lastly, omitting the extraneous interfaces can reduce development time across the entire software development life cycle. Making a decision to not implement an interface early in the development cycle reduces a cascade of requirements that span the entirety of the development process. When the decision is made to implement an interface, even a dummy implementation, it incurs additional costs such as requirements analysis, design decisions, development time, software integration and testing and compliance testing. The total effort saved as a result of not performing those activities can result in a significant time savings that will grow linearly as additional components are incorporated within the system.

3.1.6.3 SCA Solution

During the design process two approaches were considered as routes to get to the endpoint of lightweight components. The selected approach, illustrated in Figure 18, can be thought of as optional composition. In optional composition, a component would only realize the interfaces "<interface>" that it needs. In the example, the My WF Component realization would have the option of providing an implementation for either the *PropertySet* and/or the *Lifecycle* interfaces.

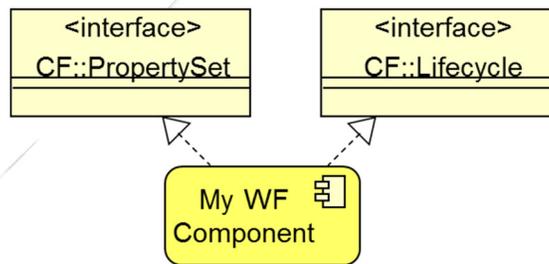


Figure 18 Component Optional Composition

3.1.6.4 Implementation Considerations

The optional composition approach comes with implications on the framework implementation which are associated with the two scenarios represented in Figure 19. In the example on the left, the framework needs to account for My WF Component having a relationship with either or both interfaces. In the other scenario a component implementation defines an implementation specific interface to act as an intermediary that combines the required interfaces into a single reference. In this case the framework cannot make any implementation decisions that preclude a developer from utilizing that type of design. Both of these are viable alternatives, and existing Core Framework

implementations may need to incorporate additional “is_a” calls within a CORBA PSM, to determine whether or not a component realizes a particular interface.

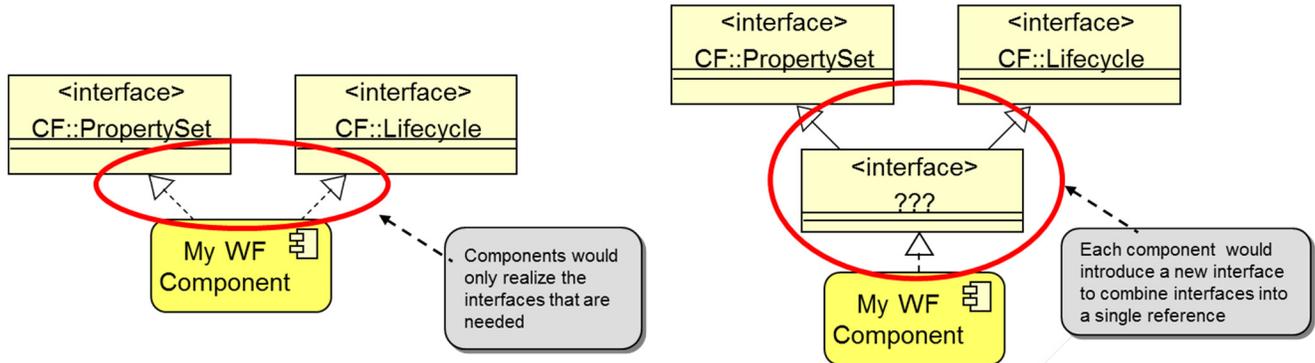


Figure 19 Optional Composition Design Approaches

Additionally, the ApplicationManagerComponent does not use any of the lightweight configurations. This constraint is included to preserve backwards compatibility with earlier implementations.

An important point to keep in consideration is that Lightweight Components are an optional capability. If a developer chooses not to leverage the optional composition capability then they are able to develop compliant applications that are very similar to those produced in accordance with SCA 2.2.2. Some developers may determine that the enhancements provided by Lightweight Components do not exceed the cost benefit threshold associated with the change. However Lightweight Components provides SCA users with a common pattern and approach to optimize components for those that would benefit from the capability.

3.1.7 Component Model

3.1.7.1 Overview

The SCA component model provides a means to improve the clarity and consistency of the specification. Earlier SCA versions contained numerous references to “components”, but did not define the term and used it very inconsistently throughout the document. Consequently, a large burden was placed on the reader to determine which elements described the attributes of runtime system elements. The presence of the component model also provides a foundation for the use of software modeling and Model Driven Development techniques within the development of SCA compliant products. Figure 20 illustrates some of the SCA components.

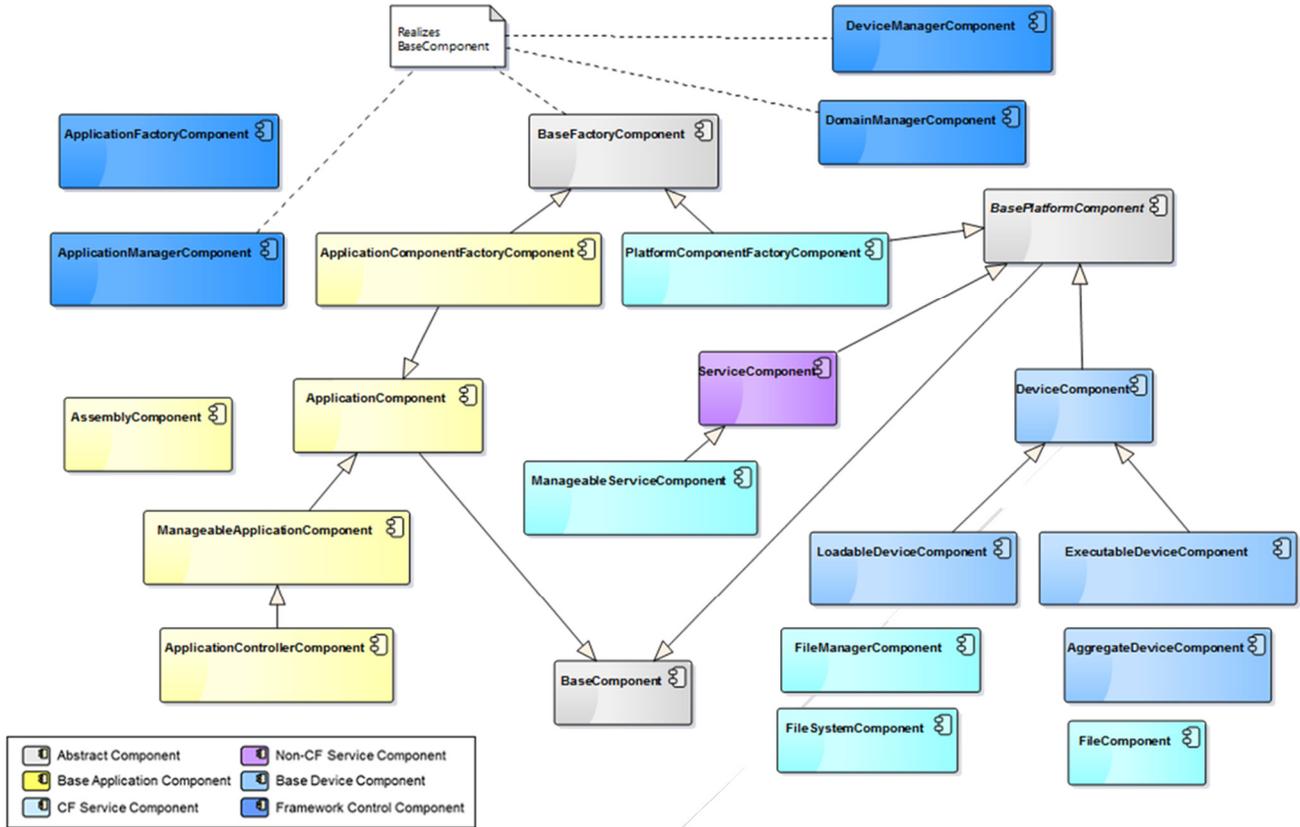


Figure 20 SCA Component Relationships

3.1.7.2 Interfaces and Components

SCA 2.2.2 was expressed in terms of interfaces, or more specifically CORBA interfaces. Accompanying each interface specification was information describing its associations, semantics and requirements. This representation of information was often challenging for new readers of the specification because it did not align with their expectations of what an interface should provide and it did not support an easy decomposition of implementation responsibilities.

An interface is a shared boundary or connection between two entities. It specifies a well-defined, and limited role which needs to be fulfilled. The role may either be functional (defined specific behavior to be performed; “to do” or non-functional (identifies criteria used to judge the qualities of operation: “to be”). Interfaces define “what” needs to be done, “why” something needs to be done, but not “how” to do it. As such, most pure interfaces tend to be stateless.

Since a well-defined interface defines a limited role, and complex system elements generally need to fulfill multiple roles, multiple, separate interfaces are often required to fully support the set of functional and non-functional requirements. It is often the case that multiple interfaces need to interact with one another and only certain sequences of those interactions will result in useful functionality. Therefore it is often useful to package these interactions between multiple interfaces into an integrated unit of defined behavior known as a component.

A Component is an autonomous unit within a system or subsystem. Components provide one or more interfaces which users may access and encapsulate the internals of how they are provided other than as accessed by their interfaces.

Components provide a modular, replaceable part of a system, which within its defined environment:

- implement a self-contained lifecycle, which may include sequential interaction requirements which exist between multiple provided interfaces
- present a complete and consistent view of its execution requirements (MIPS, memory, etc) to its physical environment
- serve as a type definition, whose conformance is defined by its 'provided' and 'required' interfaces
- encompass static and dynamic semantics

Table 2 Characteristics of Component and Interfaces

Interface Characteristic	Component Characteristic
Role -oriented → best suited as problem domain / analysis-level abstractions	Service -oriented → best suited as solution domain / functional-level abstractions
Conceptual / Abstract / Unbounded Responsibilities	Practical / Concrete / Constrained Responsibilities
Have no implementation mechanisms	Can – and often do – provide prototype or default implementations
A necessary, though not sufficient, element of Portability and Detailed Architecture / Design Reuse	Properly-developed, Components improve prospects of Portability and Detailed Architecture / Design Reuse
Interfaces are generally SYNTAX without an underlying SEMANTIC definition, and are generally seen as STATELESS as a result	Components <u>MUST HAVE</u> well-defined SEMANTIC baselines because they fulfill multiple Roles within a Framework → Components are <u>MUCH-MORE</u> than the sum of the Interfaces which they implement

3.1.7.3 Benefits and Implications

The introduction of the component model provides a concrete bridge from interface to implementation responsibilities and a well-defined path for integrating model based software engineering techniques within the development process. Having these abilities will be even more important as usage of SCA optional composition becomes more prevalent.

The textual and formatting changes associated with the incorporation of components within the framework are visually intimidating because they introduce several new sections, new model elements and relocate text. The division of responsibilities may at times look duplicative e.g. why there is a need for a *DomainManager* interface and a *DomainManagerComponent*. However, as you read the corresponding sections it will become apparent that in most cases the component oriented sections include semantics and requirements associated with deployed or executing systems or elements.

In terms of the SCA product implementation, the impact of the component model should be negligible. The component model does not contain any constructs that map into IDL, therefore any requirements that are implemented by a product developer must be done within the context of the IDL generated from the interface definitions. In fact, the layout represents how most current SCA developments already implement their software elements:

- The developer creates an implementation class that represents a component, e.g. a `ManageableApplicationComponent`
- The implementation class has associations with other classes that correspond to *CF::LifeCycle*, *PortAccessor*, *PropertySet* and other interfaces
- The implementation fulfills the roles, behaviors and interfaces prescribed by its incorporated SCA elements

The component model is still a work in progress within the specification for a couple of reasons. There were a number of modifications made to accommodate inclusion of the new concept and it is fully expected that some elements that should have been moved were not. Secondly, at time of publication, the group had not come to consensus on far reaching decisions such as whether or not exception throwing should be described in an interface or component sections.

It is expected that these and other issues related to components will continue to evolve in future revisions of the specification, however, consistent with the earlier discussions, these modifications will improve the quality of the specifications and enhance its use within modeling environments but they should have no impact on an SCA product implementation.

3.1.8 Units of Functionality and SCA Profiles

3.1.8.1 Overview

Earlier SCA versions have subscribed to a “one size fits all” approach to implementation and specification compliance. The documents described the SCA elements and associated a set of requirements with each construct. When a developer chose to incorporate an instance of one of those elements they were responsible for implementing all of the associated requirements or seeking a waiver for any capabilities that were not provided.

The SCA Units of Functionality (UOF) and Profiles were developed to address the restrictions imposed by the earlier specifications. The intent behind the UOFs was to introduce a set of flexible constructs within the framework that allowed SCA to accommodate a wide variety of target platform (e.g. resource constrained, fixed wing aircraft) and architecture (e.g. single versus multiple channel) specific requirements gracefully which in turn support the development of more “mission-focused” products.

The primary benefit associated with having UOFs is that they provide a standardized approach that allows interfaces and requirements that are not appropriate for a product to be omitted from the component specification. The elimination of these requirements has the following ancillary benefits:

- Reduced footprint – being able to omit unnecessary interfaces reduces the size of the deployed object. Even a stubbed interface realization requires a small amount of space and these small savings can add up
- Increased assurance – reducing the size of the developed object increases the degree to which the code can be assessed. The reduction in size minimizes the number of locations in the product that could be exploited. Likewise, having dead or stubbed code introduces additional locations where vulnerabilities might exist
- Reduced development time – having fewer requirements has a direct correlation with smaller projects and shorter development cycles
- Enhanced product performance – reducing object size and removing unnecessary modules improves the performance as there is less code to execute and fewer opportunities for superfluous context switches

3.1.8.2 SCA UOFs and Profiles

SCA UOFs were intended to be understood in a manner similar to their POSIX[®] namesakes: a Unit of Functionality is a subset of the larger specification that can be supported in isolation, without a system having to support the whole specification. The initial design philosophy behind UOFs was that they should be restricted to optional SCA features. However, this was relaxed as the specification matured so there are some UOFs that are associated with mandatory capabilities. Part of the rationale behind the expansion was to identify and highlight tightly coupled requirements, the other was to accommodate discussions regarding whether or not some of those capabilities might become optional in the future. Even with the expansion not all SCA requirements are associated with a UOF.

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

The Profiles comprise a set of UOFs, the collection of which is intended to be aligned with common, real world platform configurations. SCA Profiles are only applicable to OEs because it was easier to forecast a relatively small set of common configurations for distinct classes of target platforms. The profiles provide a common, easy way to select a UOF configuration of compliant SCA radios, from an almost infinitely flexible platform with the Full Profile, to a minimalist, Lightweight Profile, platform where the radio boots and begins executing a single waveform with little configuration and processing.

3.1.8.3 Use of UOFs and Profiles

Appendix F (reference [6]), similar to many of the other SCA documents, provides a couple sample conformance statements. The UOFs and Profiles provide the mechanism to align a product's design with its mission. The product developer must communicate a product's capabilities to external consumers and stakeholders. The following text represents an example conformance statement: "Product B is an SCA conformant Operating Environment (OE) in accordance with the SCA Medium Profile containing an SCA Lightweight Application Environment Profile conforming POSIX[®] layer and an SCA Full CORBA Profile transfer mechanism".

In this example the statement contains an explicit reference to a profile (Medium). Figure 21 dictates the approximately 226 requirements that are applicable requirements for this product. The Medium profile contains the Management Registration, AEP Provider and Deployment UOFs and the specific requirements are identified in the SCA Appendix F Attachment 1: SCA Conformance mapping spreadsheet.

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

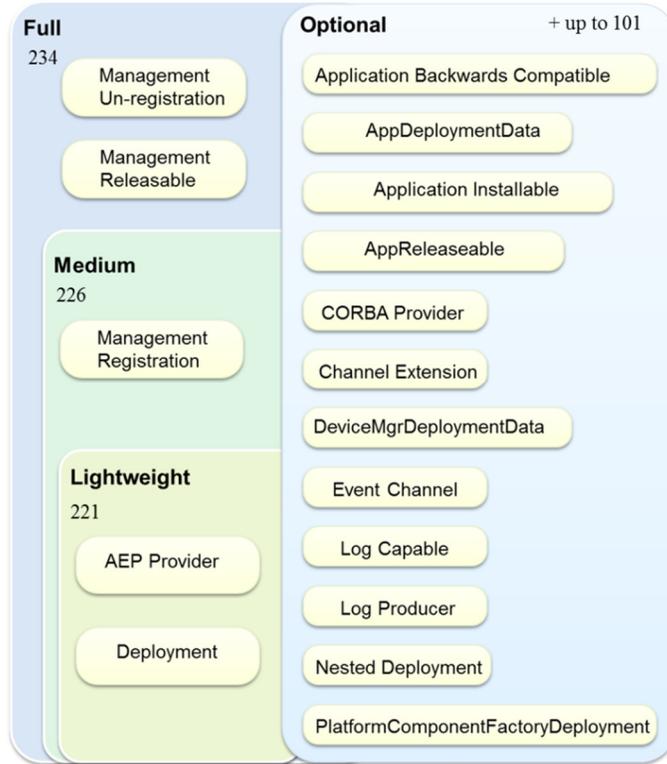


Figure 21 SCA Profiles with OE Units of Functionality

The sample conformance statement could be refined to include additional units of functionality as follows: “Product B is an SCA conformant Operating Environment (OE) in accordance with the SCA Medium Profile which contains an SCA Lightweight Application Environment Profile conforming POSIX® layer and an SCA Full CORBA Profile transfer mechanism, and extended by the Log Capable, Log Producer and Event Channel UOFs”.

The majority of the SCAs ability to be tailored resides within the optional UOFs. At the BaseComponent level 14 standardized capabilities and approximately 81 requirements exist that could be applied to a component.

The SCA was not developed with the intent of excluding a mandatory unit of functionality from a profile. The likelihood of having to do so now is unlikely as the profiles do not include that many UOFs, however the profile concept is still developing so the benefits of utilizing that type of strategy will need to be evaluated if the need arises.

3.1.9 Late Registration

Component registration is accomplished using a push model approach as described in Section 3.2. The SCA components which provide a registration capability are the ApplicationFactoryComponent, DeviceManagerComponent and DomainManagerComponent. In most instances component registration follows a standard pattern; a component registry, that is associated with a manager component, comes into existence, the manager component deploys all of

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

its components which subsequently register with their deploying component via its associated registry, the manager component takes whatever actions are necessary to finalize the registration.

3.1.9.1 Application Registration

Figure 22 illustrates what occurs when an application is deployed on a platform. After the ApplicationFactoryComponent deploys each ManageableApplicationComponent, the deployed component registers with the ApplicationFactoryComponent. Upon successful application creation, the ApplicationFactoryComponent returns an ApplicationManagerComponent which contains information describing all of the application's components. Both the values of the ApplicationFactoryComponent and the created ApplicationManagerComponent are stored within the DomainManagerComponent.

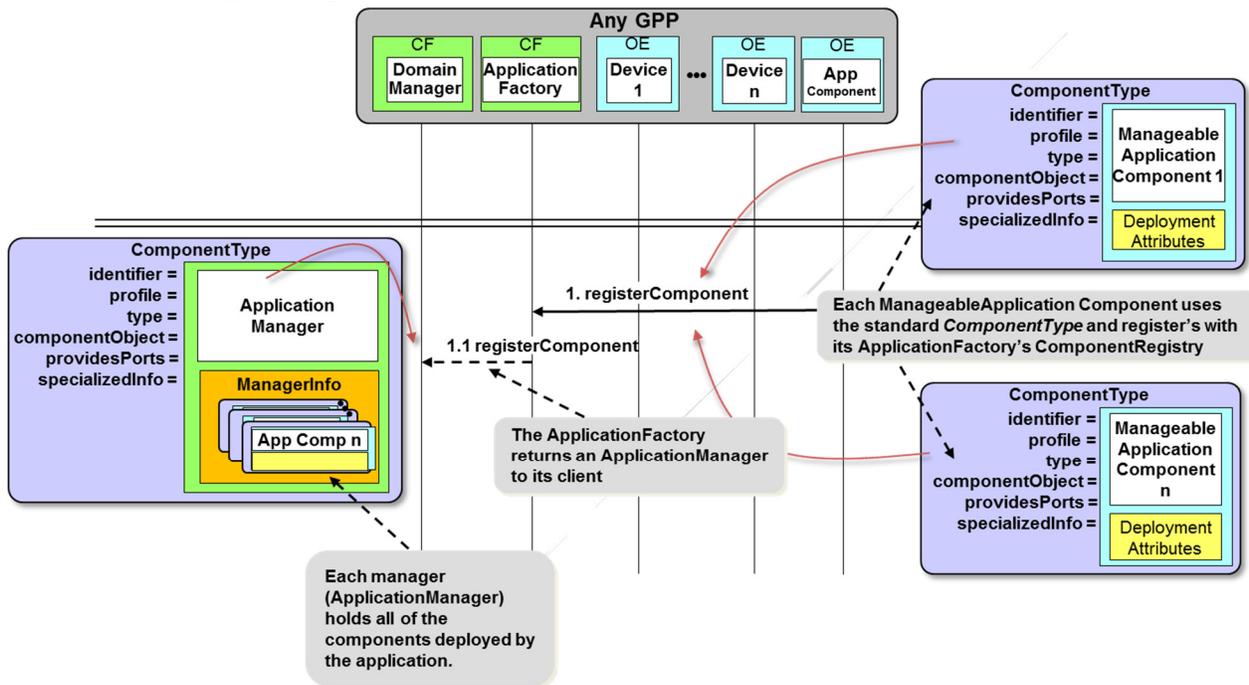


Figure 22 Application Component Registration

3.1.9.2 PlatformComponent Registration

PlatformComponent registration behaves in a similar manner as illustrated in Figure 23. Typical deployment of a PlatformComponent is initiated by a DeviceManagerComponent. As each PlatformComponent is instantiated it registers with its deploying DeviceManagerComponent. The “registration finalization” activity for PlatformComponent registration occurs when a DeviceManagerComponent registers with a DomainManagerComponent. A feature of DeviceManagerComponent registration is that in addition to registering itself, any PlatformComponents that have previously registered with the DeviceManagerComponent are registered with the DomainManagerComponent.

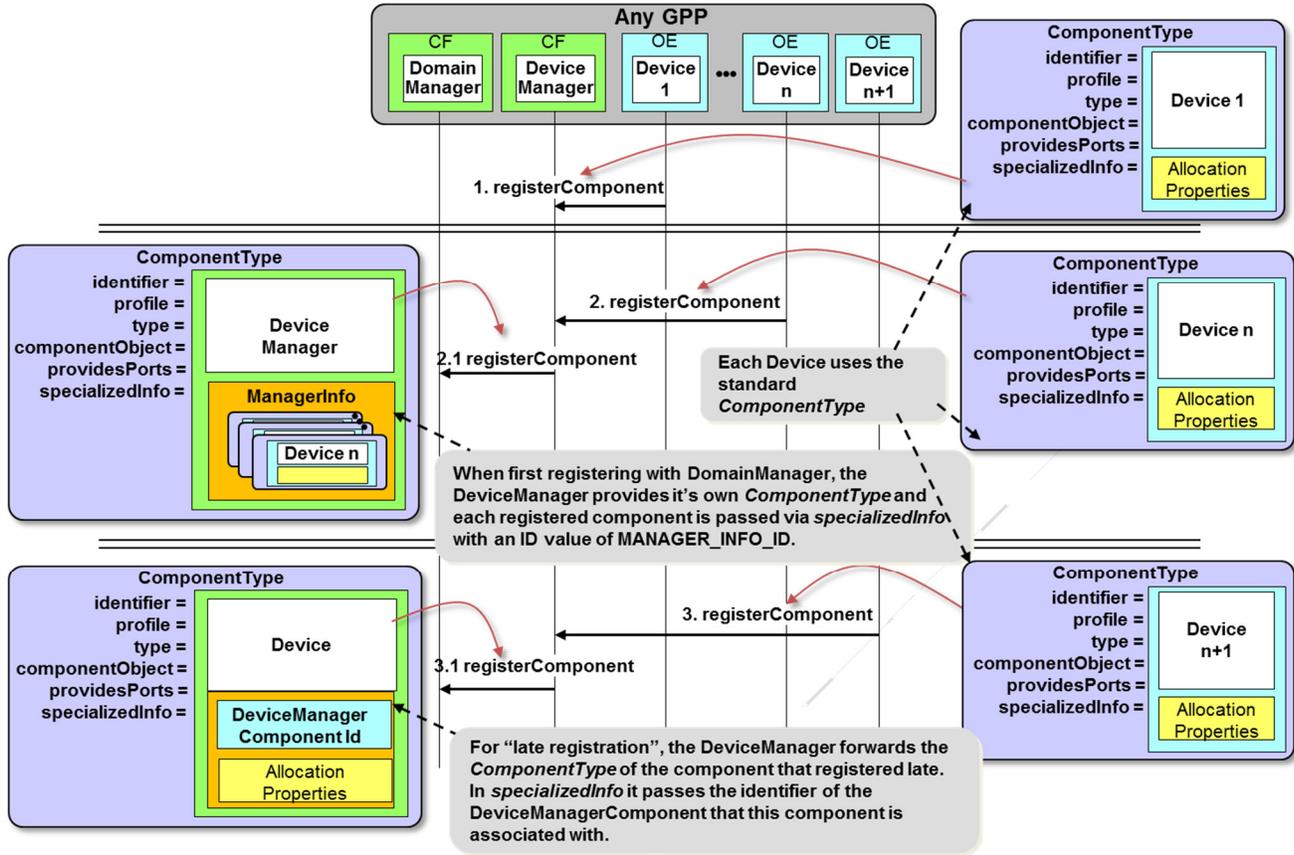


Figure 23 Platform Component Registration

3.1.9.3 Late Registration

Within SCA, late registration is defined as any PlatformComponent registration that occurs after the initial registration of its associated DeviceManagerComponent. According to this definition there are several scenarios which fall under the classification of late registration. The model scenario for late registration is associated with plug and play components that are introduced within a platform after the system has been up and running. A nuanced scenario arises in the typical deployment approach. SCA does not have a mandated time when a DeviceMangerComponent has to register so it would not be far-fetched to envision a situation where that manager would deploy 15 components and register with its DomainManagerComponent after 10 of the deployed components had registered with it. Fortunately, the design of the *ComponentRegistry* interface and the registration strategy is flexible enough that it can accommodate both scenarios. There have been suggestions that the approaches to regular and late registration are duplicative and the regular approach should be removed. The claim is a reasonable one but we have chosen not to take any action within the specification until the relative merits of both approaches have more concrete data upon which to make a determination about a way forward.

Figure 23 also illustrates the late registration scenario. In the example Device n+1 registers after the DeviceManagerComponent registered with the DomainManagerComponent. As a result, the DeviceManagerComponent implementation needs to recognize that it has already registered with the DomainManagerComponent and this is a new component registration. Once that determination

has been made, the DeviceManagerComponent is responsible for registering that component within the domain. The DeviceManagerComponent adds the registering component to its set of registered components and forwards the registration. The only modification made by the DeviceManagerComponent is the addition of an entry within the PlatformComponent's specializedInfo which indicates its associated DeviceMangerComponent; this information is used by the DomainManagerComponent as it registers the component.

3.1.10 Enhanced Process Collocation Support

Applications across all device categories continue to require better performances. Two main trends are driving the embedded device market today:

- Smaller form factors
- Improved performance per watt

However, traditional methods of achieving better performances via higher clock frequency lead to increased thermal dissipation and energy requirements. Multicore technology provides an alternative solution which improves performance per watt ratios and reduces board real-estate requirements.

This SCA release introduces support for enhanced process collocation and core affinity architectures within the framework. Core affinity is defined by its constituent parts –

- core = a complete set of registers, execution sets, etc. that are needed to execute a program
- affinity = the state of being bound to a specific logical processor

3.1.10.1 Background

POSIX[®] Operating Systems support dynamic loading of libraries and dynamic creation of threads within an OS process address space, thus allowing threads to be dynamically added to an OS process. Furthermore, today's operating systems support multi-core processors and different techniques to execute processes/threads across different cores.

Multi-core processors can operate using one of two approaches. Symmetric Multi-Processing (SMP) has a single operating system which controls more than one identical processor/core. In SMP, all processors/cores must be able to access the same memory and the same I/O devices. Multiple operating systems are used within Asymmetric Multi Processing (AMP), where one operating system exists for each processor/core. There is a great deal of flexibility within this approach as operating systems do not need to be the same and their processors/cores do not need to be identical.

Given the choice; SMP is the better alternative when communication speed between cores is critical or the workload needs to be distributed dynamically across processors or cores, while AMP is better in situations when communication speed between cores is not critical and more than one operating system is present.

Most common Operating Systems support SMP using a scheduler which allocates each task to a core while only a limited number support AMP. Real-time operating systems provide users with the ability to influence the scheduling of time-critical tasks. This ability is generally offered as part of core affinity.

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

3.1.10.2 Earlier SCA Capabilities

Prior to this extension SCA provided the ability to collocate either (all) platform or application components within the same OS process address space using a factory component but the factory was not able to create both types of components within that process. In addition, factories were somewhat static in nature as they are preconfigured with the types of components that they can create. SCA also provided limited support for multi-core devices deployment via the `ExecutableDeviceComponent`. The system designer was able to model the platform using either a single `ExecutableDeviceComponent` per core or one component for multiple cores. Using either strategy allowed the framework/OS to make the determination of where each executable should be deployed.

3.1.10.3 Enhanced SCA Capabilities

The process collocation enhancements provides support for the following features:

- executable Device Component dynamic threading.
- mixing `ApplicationComponent` and `PlatformComponent` threads within the same OS process space.
- multi-core devices deployment via core affinity requirements.

Affinity was introduced because it introduces a valuable capability within the framework, represents the most basic SMP scheduling technique and is widely supported by embedded operating systems. The proposal does not introduce any more advanced scheduling techniques because their implementations are more proprietary in nature. However, core affinity can be extended to support more complex designs such as core reservation by only allowing a single task to have an affinity for the reserved core and not allowing any of the other tasks to use that core.

Core affinity complements the existing SCA capabilities which govern component deployment. Candidate `DeviceComponents` that can host a component are selected by using any allocation properties, allocation properties, deployment channels, etc. identified by that component. After those items are evaluated and processed a target `ExecutableDeviceComponent` is selected. Any existing affinity preferences that accompany that component are then passed to the `ExecutableDeviceComponent` which, if it supports the capability, is responsible for mapping those requirements to the underlying operating system.

3.1.11 Self-Launching Components

Self-launching `PlatformComponents` are those which come into existence in a manner other than being deployed by a `DeviceManagerComponent`. These components are often associated with plug and play scenarios, however they could also be employed as part of a routine system startup. Once a platform component that will be managed by the framework is launched it is subject to all of the `PlatformComponent` requirements.

The primary issue to be addressed related to self-launching components is if/how they are associated with the framework. SCA does not dictate an approach for this situation so a system designer will need to use an implementation specific approach to associate the two components and provide an endpoint that the `PlatformComponent` can use to register. The component registry location could be provided via approach such as a property, within a designated file or as an argument to the component's executable file.

When the component registers, it is responsible for providing a `ComponentType` argument to the `registercomponent` operation. As an SCA component the self-launching `PlatformComponent` will

have a companion set of descriptor files (profile). The PlatformComponent will populate the ComponentType parameter either with information from the profile, it does not necessarily have to parse the profile, or corresponding information that it received through its execute parameters. If the PlatformComponent does not populate its allocation properties then the DeviceManagerComponent with which it registers will satisfy that requirement.

3.2 DESIGN GUIDANCE

3.2.1 CORBA profiles

3.2.1.1 Guidance on the use of Any

On systems with limited resources, the use of the OMG IDL Any data type should be minimized. The Any data type should not be used within the data path or in situations with demanding performance requirements. When an Any type must be used, it should be associated with a simple type. The CF::Properties data type is the only SCA construct that contains an Any data type within its data structure definition.

3.2.1.1.1 Rationale for restrictions on the use of Any

The Any data type should be avoided due to the significant performance and resource consumption implications that it may levy on method calls that use them. Many ORB providers supply insertion and extraction operations for known simple types and transport them without large TypeCodes that can increase message sizes significantly (in some cases the type information can more than double the size of the messages). The potential size implications are even greater for complex types, the CORBA compiler must generate code for insertion and extraction and add it to each component using the interface as well as adding the type information to each message.

The additional size and processing complexity associated with marshaling and unmarshaling utilizes resources that could be better directed towards providing application critical capabilities.

It is not necessary to find an ORB that does not support complex types in Any, or to remove the capability from a commercial product because the majority of resource savings are achieved because an Application does not use a capability, not from its absence. For example, in user defined IDL types the Any capability is turned on when the operator is generated by the IDL compiler and used by the code. However, some ORBs do have the ability to optimize for size by only including the Any capability when it is linked with an application through the use of a modular architecture.

3.2.1.2 Guidance on the availability of commercial ORBs implementing these profiles

Initially there may be few, if any, commercial ORBs available that provide an implementation tailored in accordance with the SCA specified profiles. With few noted exceptions, the Full and Lightweight CORBA profiles are proper subsets of the CORBA/e Compact profile [3]. This means that a processing element with sufficient resources could use a CORBA/e Compact ORB, support nearly all permitted Application features and require minimal porting effort.

3.2.1.3 Use Case for the Lightweight profile

The Lightweight profile is intended for extremely limited processing elements, such as most DSPs, and assumes an approach for implementing SCA components (ManageableApplication or Device) that strives to maximize performance and minimize resource utilization. In order to avoid resource intensive features of the SCA for component management, such as the

ManageableApplicationComponent's inherited *LifeCycle* interface, the Lightweight profile accommodates partially realized SCA components, Figure 24, or scenarios where the complete SCA component implementation is split between an extremely limited and a somewhat less limited processing element.

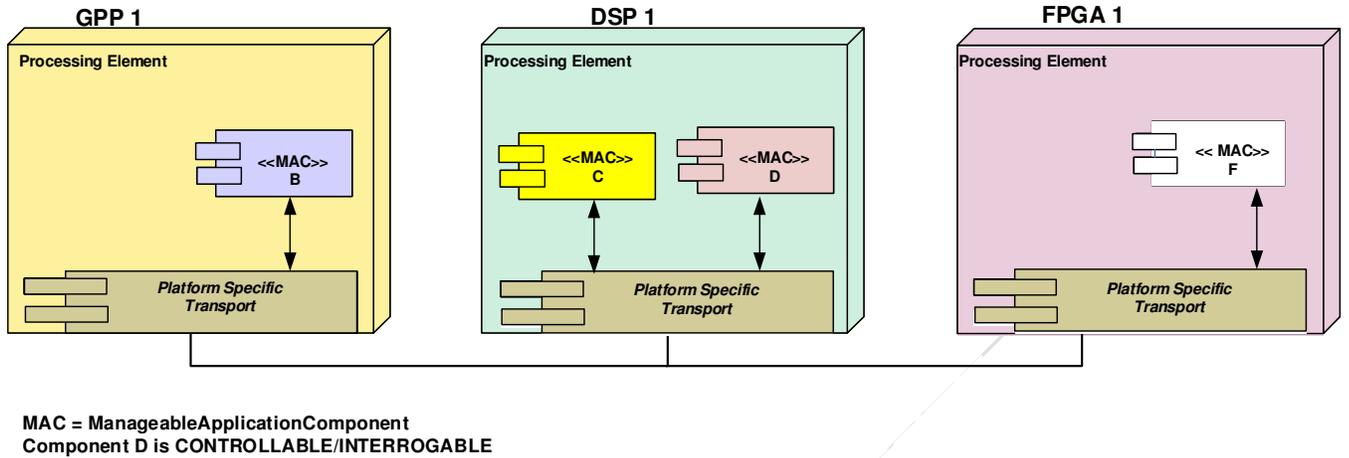


Figure 24 Lightweight Component in Lightweight profile

It is assumed that the requisite component management functions for the ManageableApplicationComponent under development are realized on the less limited processing element and only port implementations (such as traffic data handling) are realized on the limited processor, Figure 25.

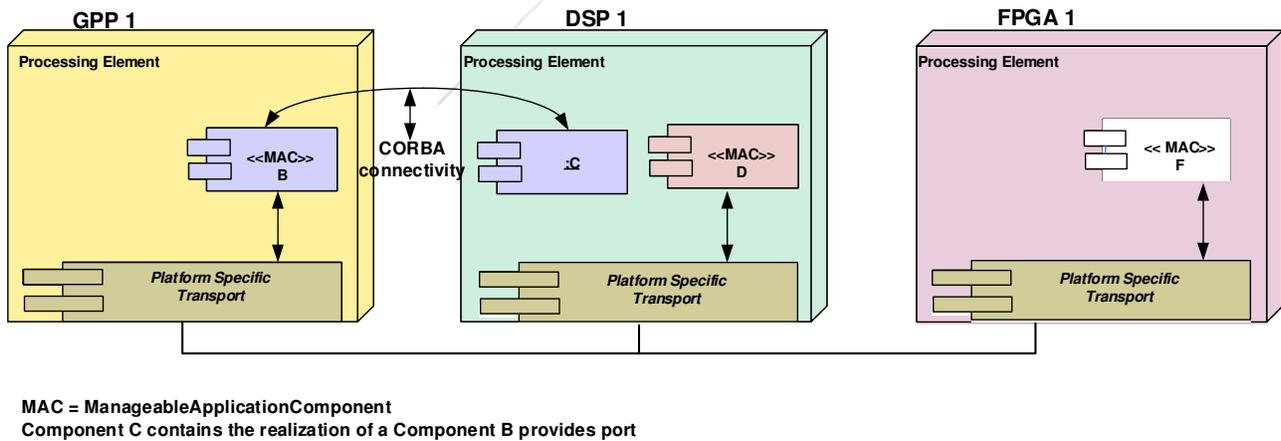


Figure 25 Component distributed across multiple processing elements

An alternative approach for applications is for an ApplicationControllerComponent to manage a component directly, i.e. not using a BaseComponent's port. In that scenario the permitted data types and method calls are restricted to those necessary for the port implementations. Note that some current standard APIs such as, Audio Port Device and GPS Device would need to be

modified to follow these constraints. Coordination between the lightweight and management portions of a component is outside the scope of this recommendation and not required to use CORBA.

Components may need to be deployed on even more limited processors such as FPGAs or have interfaces to other components on such processors, Figure 26.

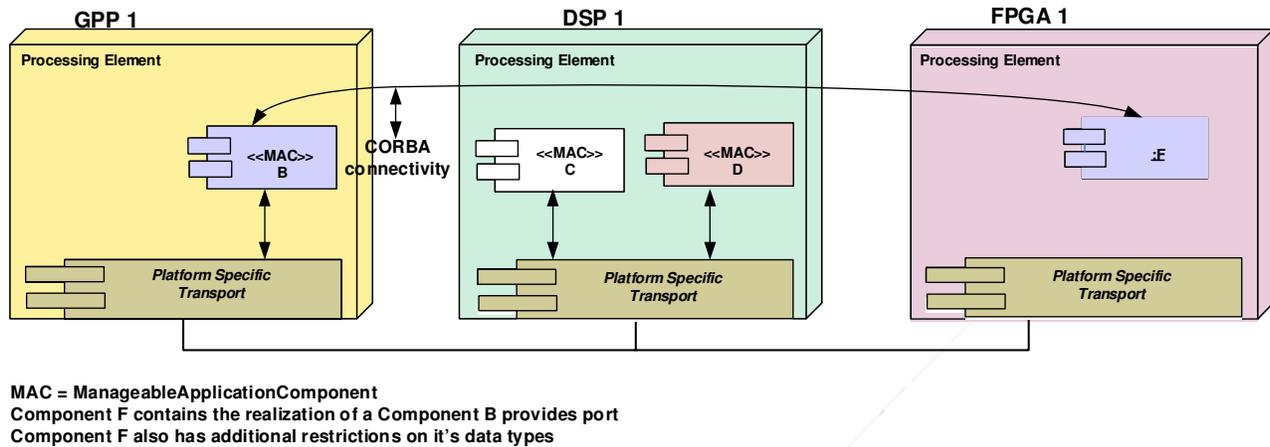


Figure 26 Distributed component with FPGA portion

Compatibility will be enhanced in these instances if data types are restricted to those realizable on such processors. Therefore, components implementing the lightweight profile are encouraged to avoid using the data types discouraged and marked with * in the table of Attachment 1 to Appendix E-2 (see reference [4]).

3.2.1.4 Guidance on restriction interface data types

It is recommended that data types be restricted in any interface to modules implemented on extremely limited processing elements such as FPGAs and most DSPs.

Interfaces to code modules implemented on extremely limited processing elements, such as FPGAs and most DSPs, whether or not they are implemented in CORBA, are encouraged to refrain from using the data types marked with * in the Lightweight CORBA profile.

This recommendation is intended to enhance portability of CORBA to non-CORBA implementations and to ensure that data can be exchanged easily between CORBA and non-CORBA components.

3.2.1.5 Rationale for CORBA feature inclusion in the profiles

The choice to include CORBA features in the profiles was driven by use cases. Some of these use cases are listed along with columns comparing Full with minimumCORBA and CORBA/e Compact in Attachment 1 to Appendix E-2 (see reference [4]).

3.2.2 SCA Waveform Construction

3.2.2.1 Overview

The SCA component structure contains a collection of building blocks that a product developer can combine in order to produce a deliverable, e.g. a waveform or service implementation. The process of creating an end product requires a series of engineering decisions, which from an SCA perspective are centered on decomposing the overall product functionality into encapsulated elements that can be integrated with the defined SCA components.

3.2.2.2 FM3TR waveform example

The publicly available FM3TR waveform architecture is illustrated in Figure 27 (this waveform is available from the JTNC Open Source Information Repository [13]). The yellow-colored components represent radio set functionality, whereas the red and blue colored blocks represent waveform software components.

SCA contains component definitions that should be used for each macro-sized component. Any of the macro-sized waveform components, for example the Data Link Control (DLC) component, could be implemented by aggregating several smaller modules or routines, but those routines would be bundled and it would only expose functionality to external users via a consolidated set of interfaces.

SCA utilizes a “port” construct as the mechanism by which a component may be extended to provide application specific functionality and behavior. The blue and red ManageableApplicationComponents on the GPP expose: in, out, and control ports. The core framework can connect the *port* interfaces to other ApplicationComponents or BasePlatformComponents in order to provide overall waveform functionality. Generally, the ‘in’ ports are described as ‘provides’ ports, whereas the ‘out’ ports are ‘uses’ ports, because they either provide or use port connections, respectively.

Using either the middleware services provided by the radio set, or direct C++ pointers, connection IDs and object references permit independent software components to communicate. The components only need each other's pointer or object reference. The messaging becomes more difficult if the components are distributed into separate memory partitions. For such deployments, middleware services provide a general solution to be applied throughout the complete radio set.

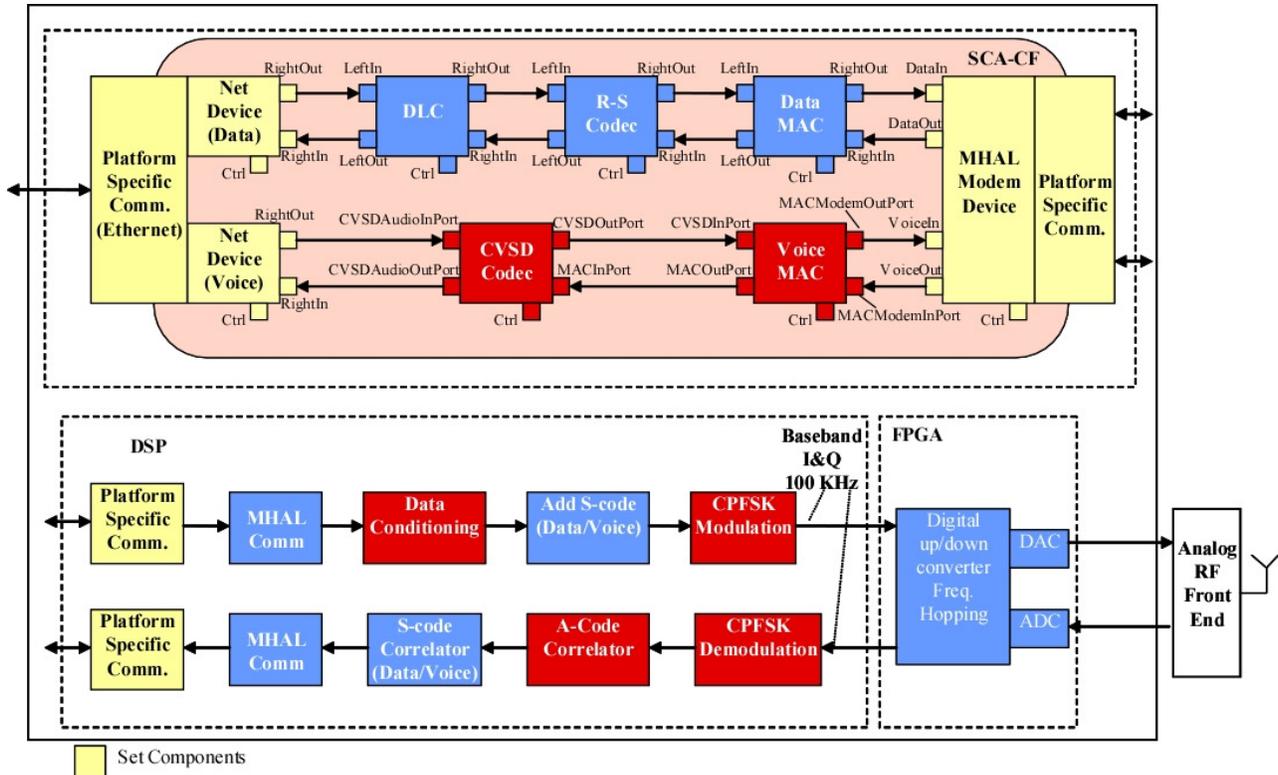


Figure 27 Example FM3TR SCA Waveform Design

The FM3TR waveform is a simple time domain multiplexed access (TDMA) application with Continuous Phase Frequency Shift Keying (CPFSK) as the baseband modulation. The JTNC implementation provides either data or voice operation. Continuously Variable-Slope Delta modulation (CVSD) is implemented for the vocoder. Reed-Solomon (R-S) forward error coding is used to improve the bit reliability of the wireless link.

The Data Multiple Access Control (MAC) is an SCA ManageableApplicationComponent that converts the input data stream into data symbols grouped to match the R-S coding format. The voice MAC performs a similar operation for the data stream produced by the vocoder. The A-code is a simple 32-bit synchronization code used to synchronize transmitter and receiver. The S-code is a second synchronization word used to identify data packet types such as voice, data, etc.

The architecture and deployment of this waveform is fairly typical for SCA implementations, although other variations are possible. In this example, the waveform components deployed on the FPGA and DSP do not have SCA interfaces. Historically radio architects have attempted to wring the last drop of performance from the DSP and FPGA devices and not implemented SCA interfaces on these lower-level software components. There is a substantial cost for this strategy – a loss of portability for these waveform components. However, advances have made extending the full SCA model beyond the bounds of the GPP much more technically feasible.

An example logical model of an FM3TR radio is illustrated in Figure 28, complete with radio devices, services, and core framework components.

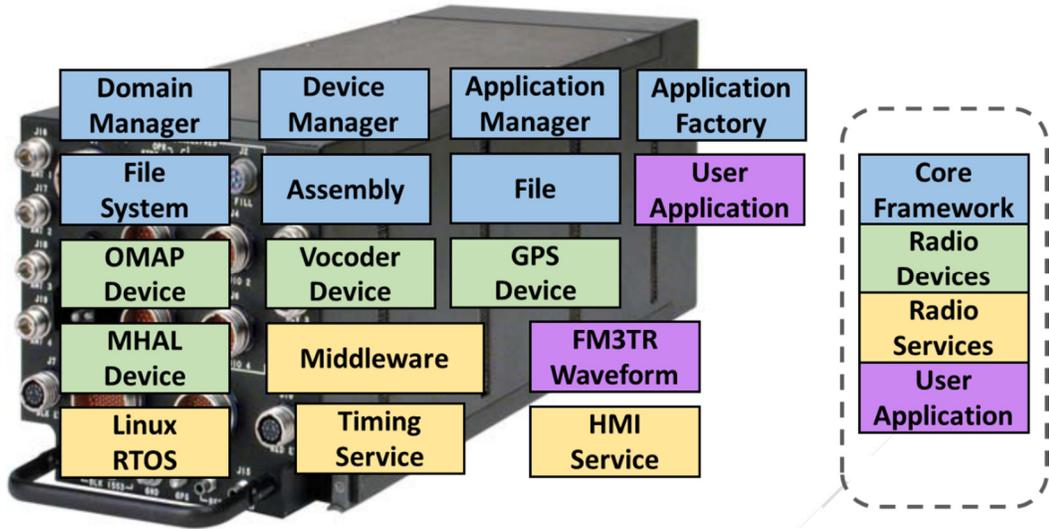


Figure 28 Example Deployment of FM3TR

3.2.3 Static Deployment

3.2.3.1 Overview

The earlier approach to SCA deployment uses a strategy that emphasizes the framework's dynamic capabilities. Within the deployment model the ApplicationFactoryComponent creates software components by sending instructions to DeviceComponents representing the processors. After the components have been instantiated, the ApplicationFactoryComponent sends "connect" commands to the components, providing them with the object references necessary for communication with the desired component. The ApplicationFactoryComponent then reads the Software Assembly Descriptor (SAD) file to 'wire' the waveform together.

The deployment strategy is very flexible and is well suited to scenarios featuring target platforms that need to accommodate a wide breath of candidate architectures. On the flip side, the flexibility comes at a price because deployment performance (i.e. speed) can suffer when there are multiple permutations of devices and configurations that can host an application. SCA has been extended to provide additional guidance regarding how to improve deployment performance. One improvement is the deployment optimizations, a second introduces language that allows a platform to preprocess its domain profile files, thus reducing the need for xml parsing or processing to occur as part of deployment. This SCA release provides yet another optimization with the introduction of a common approach for static deployment.

3.2.3.2 Deployment Background

Figure 29 illustrates the steps that need to take place for application deployment to occur.

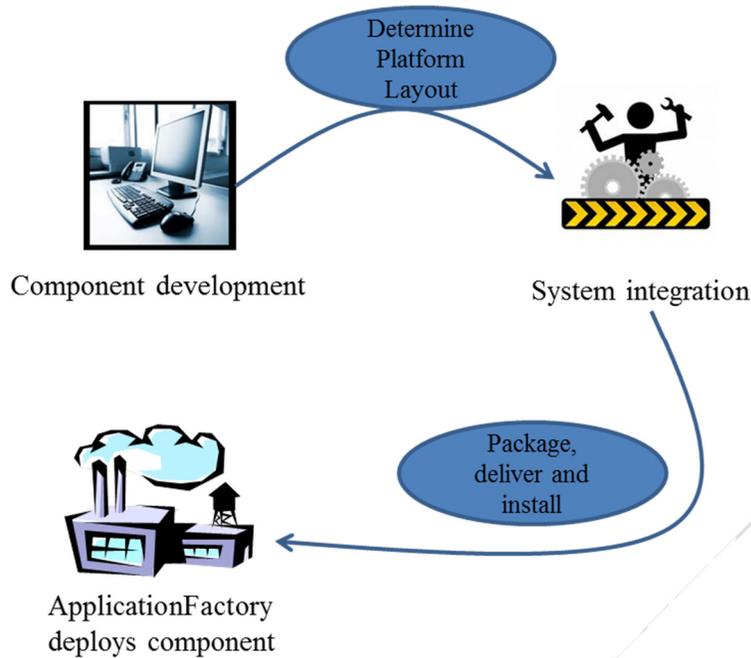


Figure 29 ApplicationFactory Role in Component Deployment

1. Developer creates individual system components
2. Platform engineers and developers identify system configuration
3. Platform provider integrates system
4. Platform provider packages and delivers product
5. Platform user / administrator deploys application
6. User uses application

Static application deployment is characterized by the framework not having to make deployment decisions or receive assistance during the process of establishing connections between `ApplicationComponents`. Having limited or no responsibilities during either of those activities expedites the deployment process because the number of decisions the framework needs to make during application instantiation are minimized.

3.2.3.3 Connection Management

SCA permits legacy (i.e. pre-SCA 4.1) connections to be established within a platform. This is accomplished by having an `ApplicationFactoryComponent` query each `ManageableApplicationComponent` for its provides port connection IDs and then send those IDs to the components that require the connection endpoint. While this is similar to the earlier SCA connection mechanism, it requires a slight modification of a legacy application. A second alternative has components return their connection IDs upon registration, thus eliminating the communication traffic required by `getProvidedPorts()`. This method is not as flexible as the first and does not support plug and play components, but it improves waveform startup times. A third approach could be employed in a more static scenario where an `ApplicationFactorComponent` receives connection information generated at build time from the domain profile files. Within this scenario, the `ApplicationFactoryComponent` would not require registration from the deployed

components because the target configuration would already be known. Fully realized, this approach would result in pre-wired applications that are ready for operation upon instantiation.

3.2.3.4 Example

This example usage of static configuration is subject to the following constraints:

1. The application does not utilize the enhanced deployment capabilities
2. The application does not create any of its components via an `ApplicationComponentFactoryComponent`

Application installation is identical to how it has always been executed, with the objective of transferring application software onto the platform. The application uses the platform capacity management mechanism and model with the assumption that the application to be deployed will fit on the desired target processing element. The application uses the `ApplicationFactory::create` operation deviceAssignments parameter, the value needs to be provided by the system developer, to target a `ManageableApplicationComponent` to a specific `DeviceComponent` (eliminating the need for the `ApplicationFactoryComponent` to determine where to deploy the component). Using a variation of the third connection approach described in the previous section, the developer will populate the SAD with a value in the `providesport` element's `stringifiedobjectref` attribute. This value implies that the `ApplicationFactoryComponent` will know the provides port location. (Note: A determination was made that given the existence of the aggregated `connectUsesPorts` operation there was not a significant improvement that would be realized by adding a static capability to supply uses port information).

A fully static approach which would eliminate the need to call the deployment machinery would require uses port information to be integrated within the deployed component as well. However, the current thinking is that any potential performance improvements associated with that approach are outweighed by its lack of flexibility.

3.2.4 Application PIM Profiles Conformance Benefits

SCA Appendix E-1 specifies profiles for SCA to use which enable application Platform Independent Modeling (PIM). The profiles identify a set of Object Management Group (OMG) IDL features that are available for use in the definition of application specific interfaces. Technically speaking, the definition of application APIs is beyond the bounds of the SCA, but it is an important topic for the specification to address when looking at the broader relationship of individual elements within an SCA compliant implementation and the degree to which they contribute to a system meeting the original JTRS objectives.

3.2.4.1 Application Conformance

Application conformance with one of the applicable IDL Profile enhances the ability of an application's design to be "loosely coupled" to its implementation. While there are many factors that contribute to portability, the incorporation of the guidance provided within the specification can increase portability significantly.

The generic definition of Portability is "the quality or state of being portable" where portable is "easy to carry or move around". The definition is applied within Computer Science as "a characteristic attributed to a computer program if it can be used in an operating systems other than the one in which it was created without requiring major rework".

While it is not explicitly stated, it could be inferred that SCA enhances portability within a somewhat homogeneous environment. When applications are designed in accordance with IDL

profiles, their underlying designs become more portable because they are better able to reside in heterogeneous environments, where there are multiple processor families and/or different programming languages.

Designing applications in accordance with the IDL profiles enables an application to be moved to a different processor environment or a subcomponent to be moved to or replaced by a different implementation within an alternative target environment, while maintaining the integrity of the application's design.

3.2.4.2 Engineering Tool Conformance

The benefits of using engineering tools within a software development process are well known and understood. The primary benefits are related to productivity and quality. Tools can perform code generation and relieve developers from having to perform some of the rote tasks related to the process of moving from the design to implementation phase of a project. This automation not only performs the step quickly but the code is less error prone and many products have been tuned to optimize the performance of the generated code.

The existence of tools can be particularly useful for organizations that are in the early stages of initiating development activities in a new environment. Tool vendors have developed expertise in the tools and techniques available for all of their target environment and they are able to share that knowledge to jumpstart an organization.

Tools can also enforce compliance with the specification. Ensuring that the developed products are compliant with the specification will allow them to fully take advantage of the associated benefits.

3.2.5 IDL PSM Constraints

Within the IDL to Language Specific Mappings section of the SCA Platform Specific Model – Language Specific Mapping [add reference] appendix, several constraints related to the Standard IDL to language mappings. The first constraint is that “The OMG C and CPP IDL to language mappings generate language elements within the CORBA name space.”. While this is true, there are strategies that a developer may use that would allow them to use Standards compliant tools and still be able to develop code that maximizes reusability and is platform independent such as the following:

- avoid IDL features such as Any or object which have a high correlation with CORBA specific features.
- modify any structures within the IDL generated code that use the CORBA name space to use the SCA CF::Primitive and Primitive Seq Types.

Taking a few simple actions such as these will permit a product developer to produce SCA compliant, language PSM artifacts quickly and correctly using commercial/open source products.

3.2.6 Organization Specific SCA Tailoring

SCA provides a framework and within that framework there are numerous ways to combine the framework elements and have an end result that aligns completely with the specification's structural and semantic requirements. The introduction of Lightweight Components within the specification was beneficial because it provided a compliant approach for SCA interfaces to be combined to develop products that are highly aligned with a specific mission. The drawback of that approach is that it no longer has a uniform set of interfaces. Figure 30 illustrates the definition of a DeviceComponent.

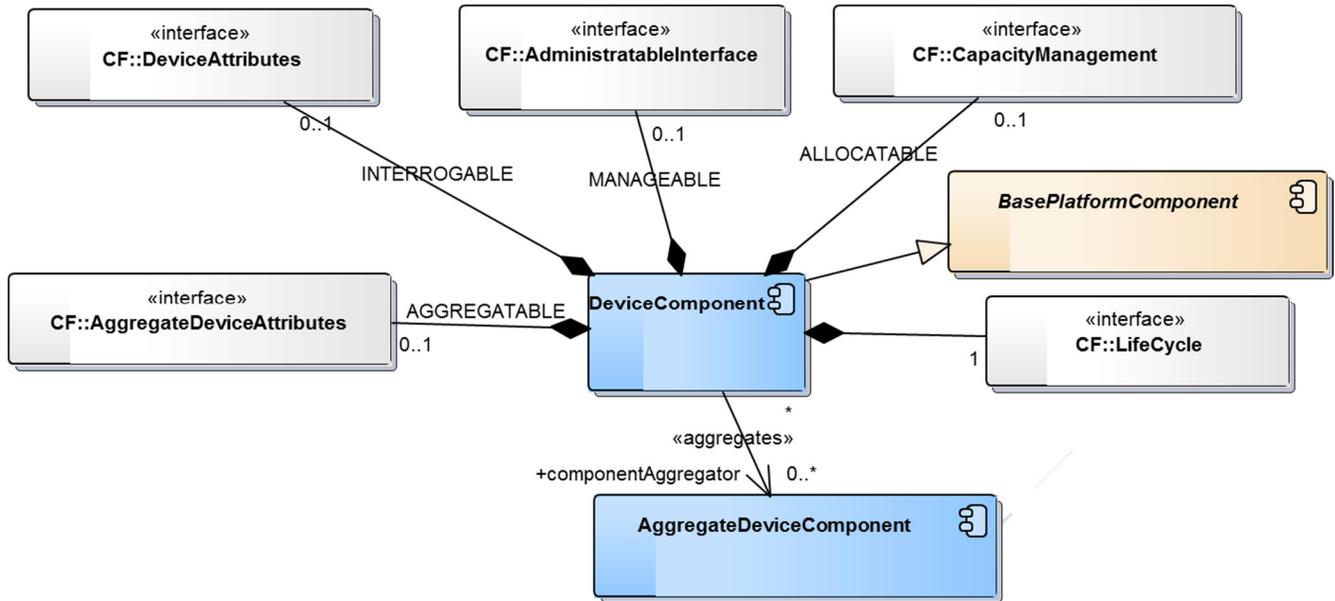


Figure 30 Device Component Definition

An organization could decide to use the SCA defined component definition as a starting point. After analysis they might determine that only the DeviceComponent level UOFs (i.e. AGGREGATABLE, INTERROGABLE, MANAGEABLE, and ALLOCATABLE) were needed within their implementation. Consequently, the following IDL interface would be used within their system:

```

interface MyOrganizationDevice : DeviceAttributes,
AdministratableInterface, CapacityManagement,
AggregateDeviceAttributes, Lifecycle
    
```

When that component that realizes the *MyOrganizationDevice* interface is integrated into a platform, its executable will be deployed by a *DeviceManagerComponent* and the resulting *DeviceComponent* will register with that manager. The Core Framework narrows the object reference provided in the registered *ComponentType* data structure to obtain information from the component. The following code snippet,

```

ComponentType myCoreFramework,
myCoreFramework.componentObject.allocateCapacity(...);
    
```

is representative of an approach that could be used to allocate capacity on this device as part of the *ApplicationFactory* application component deployment process.

This is a perfectly functional, compliant approach to take, but the composition of the *MyOrganizationDevice* interface cannot be known ahead of time by a (general purpose) Core Framework implementation, and without careful coordination the exact structure and content of that interface will not even be known within the organization.

3.2.6.1 Organization Specific Interfaces

In instances where the organization, community of interest, domain specific group, etc. knows that it will be using a common definition, they could decide to define a layer of interface definitions which resides above the SCA defined constructs. Figure 31 shows a representation of the *OrganizationDeviceInterface* interface that could be defined and maintained within the

organizational group boundary. Similar to the prior example, this interface would have associated metadata that stated that *OrganizationDeviceInterface* supports the DeviceComponent level UOFs (i.e. AGGREGATABLE, INTERROGABLE, MANAGEABLE, and ALLOCATABLE).

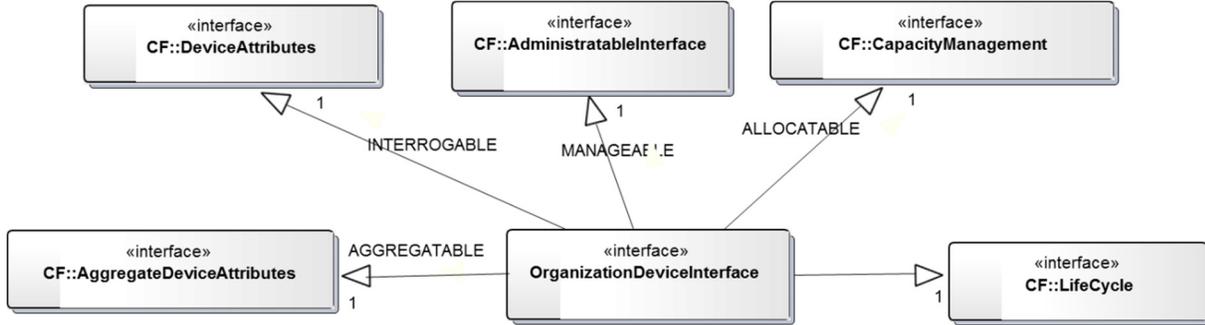


Figure 31 Definition of an Organization Specific Interface

The following code represents the *OrganizationDeviceInterface* IDL.

```
interface OrganizationDeviceInterface : DeviceAttributes,
AdministratableInterface, CapacityManagement,
AggregateDeviceAttributes, Lifecycle
```

3.2.6.2 Organization Specific Components

Our example organization might also decide to define a set of organization specific components. The organization could use their own interface and the corresponding model would look similar to that in Figure 32.

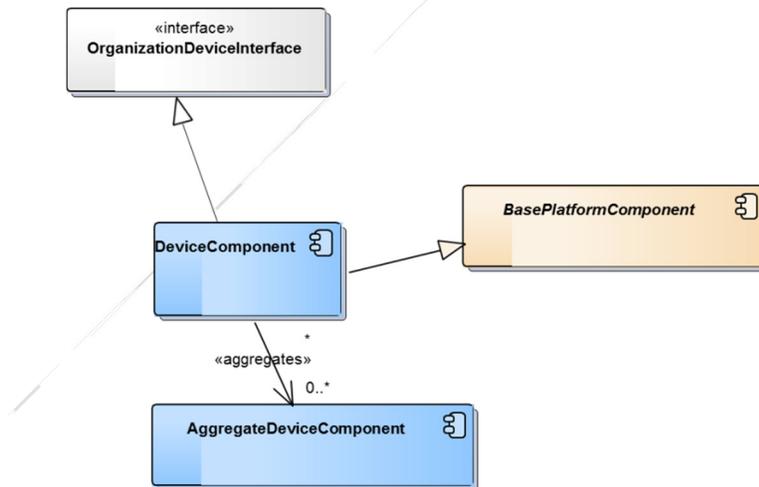


Figure 32 Use of an Organization Specific Interface

As a byproduct of the incorporation of the *OrganizationDeviceInterface* interface, the component used in the implementation would still represent the DeviceComponent level UOFs (i.e. AGGREGATABLE, INTERROGABLE, MANAGEABLE, and ALLOCATABLE) but the IDL would have a slightly different appearance

```
interface MyOrganizationDevice : OrganizationDeviceInterface,
Lifecycle
```

3.2.6.3 Organization Specific Components - Alternatives

The pattern presented by the previous examples could be reused and extended almost infinitely. If a basic set of rules and relationships are followed then all of these permutations would result in the development of component vocabularies that could be standardized within the bounds of a "community" yet remain SCA compliant. As a final example, recall the SCA BaseComponent definition.

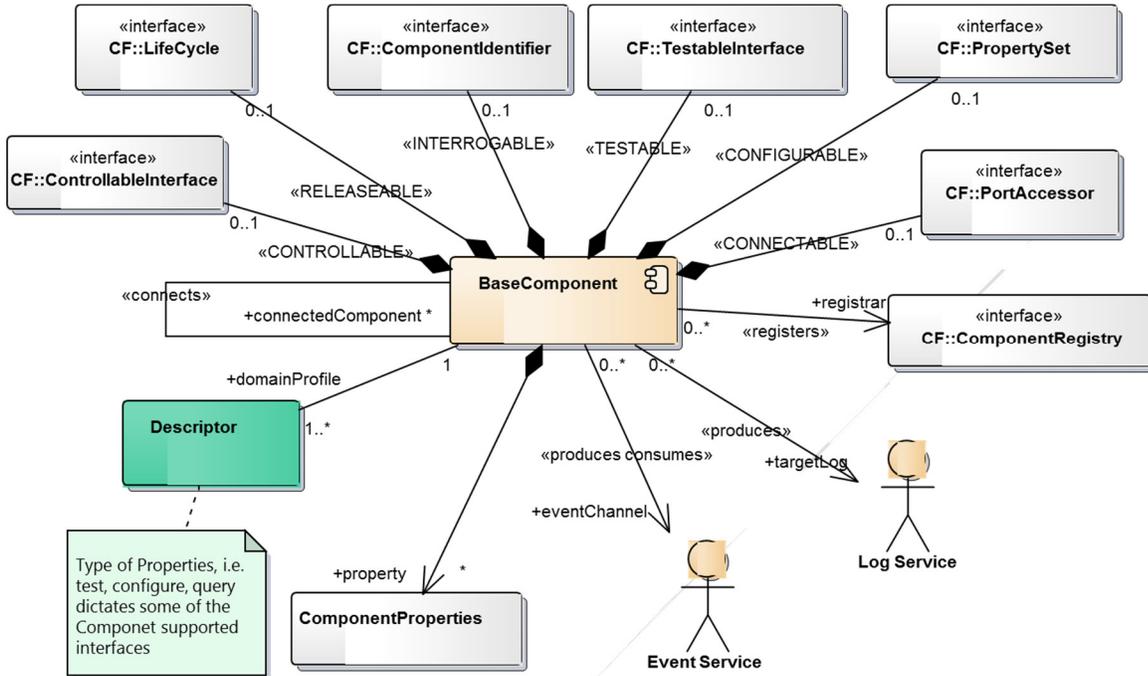


Figure 33 Base Component Definition

Also, recall that DeviceComponent has an inheritance relationship with a BaseComponent via BasePlatformComponent. Using the techniques described earlier in this section, an organization could describe their own collection of interface and component definitions and combine them together using a model similar to the one in Figure 34.

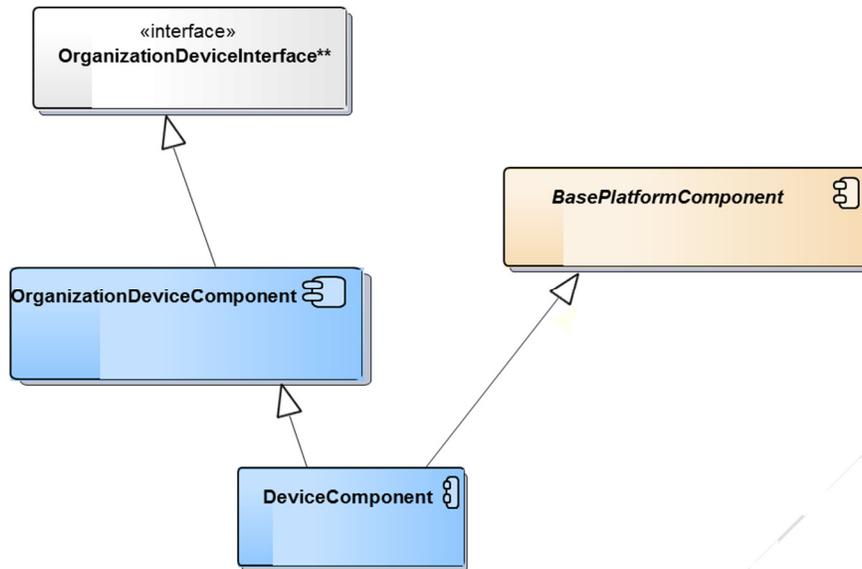


Figure 34 Model of an Organization Specific Component

If the group determined that they wanted the component to support the following UOFs: MANAGEABLE, CONTROLLABLE, CONFIGURALBE, CONNECTABLE – (i.e. not AGGREGATABLE, INTERROGABLE (DeviceComponent), ALLOCATABLE, RELEASABLE, INTERROGABLE (BaseComponent), or TESTABLE). They would implement/define a set of interfaces with a representation of

```
interface MyOrganizationDevice2 : ControllableInterface,
AdministratableInterface, PropertySet, PortAccessor, Lifecycle
or
```

```
interface MyOrganizationDevice2 OrganizationDeviceInterface**,
ControllableInterface, PropertySet, PortAccessor
```

3.2.6.4 Summary

The optional composition pattern employed by SCA Lightweight Components represents a vast departure from the interface definition approach used in SCA 2.2.2. SCA no longer has a singular set of interfaces that applications (truly all components) must adhere to. This shift places an additional responsibility on a Core Framework implementation since it is no longer able to make assumptions regarding the high level interfaces supported by a component or the set of interfaces realized by any given component. However, on the flip side it provides component developers with an additional tool that can be employed to define mission specific interfaces to be employed within a development project.

The examples provided within this section provide just a few techniques that could be used within a company / department / problem domain / enterprise / ... to introduce more uniformity of implementations or underlying components or interfaces. The structure and content are such that they can comfortably accommodate SCA compliant, layered interpretations which could at some point evolve into official or de-facto standards.

3.2.7 Sample Waveform Architecture and Considerations

The publicly available APCO-25 waveform [13] provides a representative example architecture for a simple waveform. This waveform was developed by the California Institute for Telecommunications and Information Technology (CALIT2) with support from Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS) through SPAWAR Systems Center Pacific.

APCO-25 is a suite of standards to provide interoperable digital radio communications between North American federal, state/province and local public safety agencies. The project specifies a narrowband waveform with two phases of Vocoder and Channel access scheme implementation approaches. Phase 1 waveforms use a 12.5 kHz bandwidth channel, with Frequency Division Multiple Access (FDMA) access methods and the Improved Multi-Band Excitation (IMBE) voice codec. Phase 2 uses a 6.25 kHz bandwidth channel with a 2-slot Time Division Multiple Access (TDMA) access scheme and the Advanced Multiband Excitation (AMBE)+2 voice codec for a reduced bitrate. APCO-25 also supports secure communications through the use of encryption, key management and equipment authentication.

The CALIT2 Encryption Framework [14] describes the high level platform and waveform architecture, shown in Figure 35.

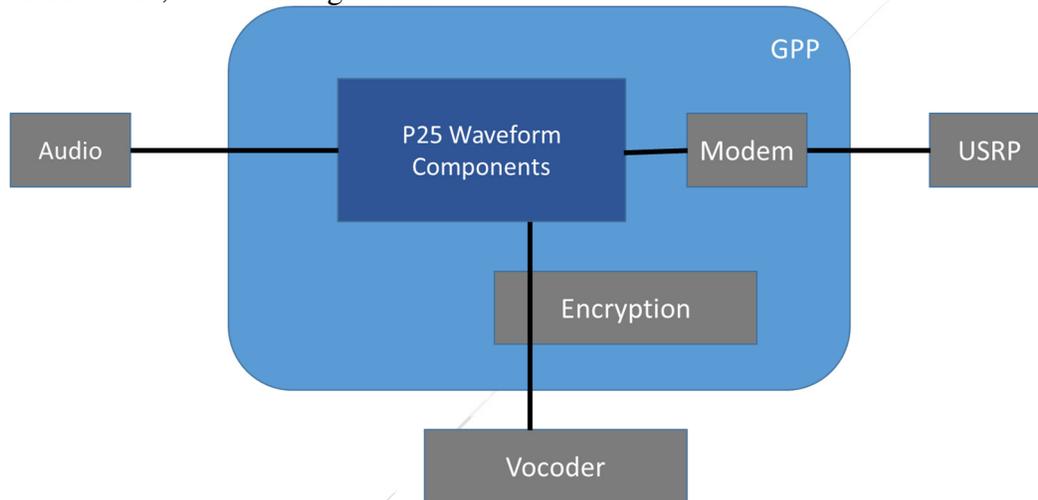


Figure 35 High Level APCO-25 Architecture

The waveform components (dark blue) assemble and extract P25 frames and directs the output to the appropriate output device be that and audio output, graphical user interface or antenna.

The Platform Infrastructure elements (grey) provide a variety of functions:

- the Audio Device captures inputs through the microphone and plays them through a speaker.
- the Vocoder provides voice encoding and decoding as well as additional features such as DTMF, single tone detection and voice activity detection. Forward Error Correction (FEC) encoding is applied to the encoded speech frames and FEC decoding is applied to the received, FEC encoded speech frames before decoding and synthesizing.
- the Modem is software-based implementation of Continuous 4 level Frequency Modulation (C4FM) consisting of a bits-to-symbol conversion module and a modulator/demodulator module.

- the Encryption Device changes information from one form to another in an attempt to hide, and then restore, its meaning. Within data communications it transforms raw data to cipher text data in order to make it unintelligible to unauthorized persons.
- the Universal Software Radio Peripheral (USRP) is a Software Defined Radio provided by Ettus Research. The USRP provides all of the basic components, e.g. ADC, DAC, that are required for baseband processing of signals. the bulk of the Core Framework implementation (not shown in the figure) resides on the GPP.

Figure 36 represents the collection and connection of SCA components required to implement the architecture.

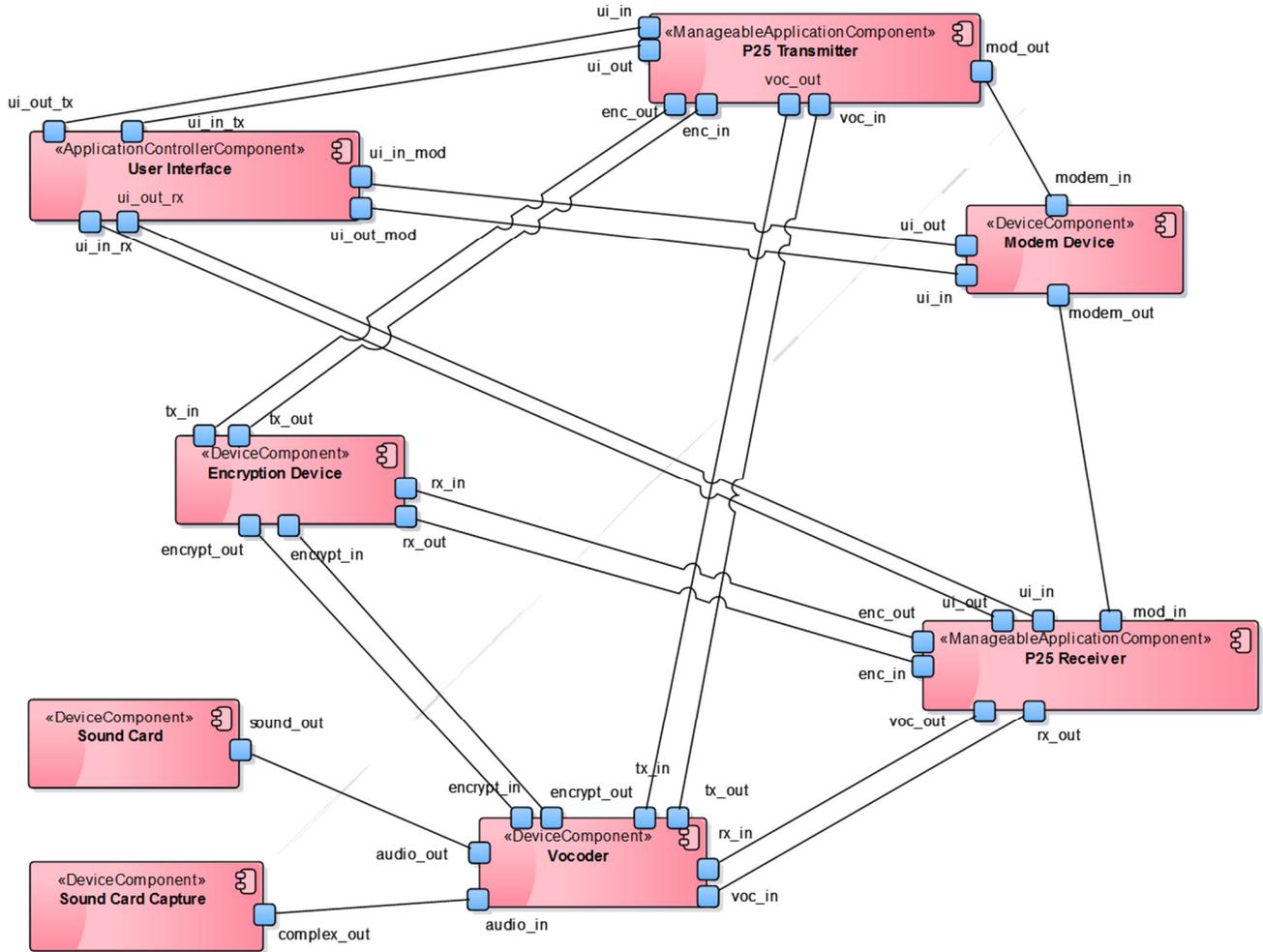


Figure 36 APCO-25 Platform Components

The interaction between the components are described by the following scenarios:

Transmit Scenario

1. The radio's User Interface (Java GUI) initiates voice communications via the Push to Talk button
2. A control packet is sent to activate the Vocoder to accept data from the Audio Device
3. The Vocoder generates FEC encoded Link Control (LC) information using data from the control packet.

5. The Sound Card sends audio data to the Vocoder
6. The Vocoder encodes and performs FEC
7. The Vocoder sends the data to the P25 Transmitter
8. The P25 Transmitter assembles the data into frames along with the LC and other control information
9. The frames are sent to the modem for baseband processing and modulation
10. The modem sends the processed information to the USRP for transmission

Receive Scenario

1. The USRP downconverts the received signal from the carrier frequency and sends it to the modem
2. The Modem resamples, demodulates and decodes the data
3. The Modem sends the decoded P25 frames to the P25 Receiver
4. The P25 Receiver extracts the LC, other control information and the voice data.
5. The P25 Receiver sends the voice data is sent to the Vocoder
6. The Vocoder decodes the data
7. The Vocoder sends the audio data to the Sound Card for playback
8. The LC and other control information are extracted from the P25 frame are sent to the Java GUI

The execution scenarios are similar when encryption is requested from the GUI. In that case, control information is sent to the Vocoder components to inform them of the encryption request. As data is processed, the AMBE encoded voice data is sent to the Encryption Device prior to FEC encoding during transmission and the encrypted AMBE encoded voice data is sent to the Encryption Device for decryption before AMBE decoding during reception. The P25 Transmitter and P25 Receiver also interact with the Encryption Device. These components exchange control data for encryption or decryption, as appropriate, during the data transmission and reception processes.

Of course prior to waveform execution, the Core Framework components interact with one another to deploy and configure the Application and Platform Components. The DeviceManagerComponent deploys the Platform Components and the Application Components are deployed by ApplicationFactoryComponent(s). The DeviceManagerComponent's Device Configuration Descriptor (DCD) identifies, deploys, configures and connects the Platform Components that it manages. The DomainManagerComponent may provide a capability to install applications. Application installation results in the construction of an ApplicationFactoryComponent which in turn is the mechanism to create, instantiate, an application. Each Application has an associated Software Assembly Descriptor (SAD) which describes its components, connections and attributes.

Each Platform and Application Component will have an associated Software Package Descriptor (SPD) and referenced domain profile files. SCA was designed to promote the development of portable applications, so an application's domain profile is largely independent from Platform Components. Keeping with the SCA philosophy, any dependencies between the elements are communicated via well-defined constructs, e.g. the SPD *usesdevice* element which provides a mechanism to specify a device in the system that supplies a capacity or required capability or the SAD *connection* element which allows an application to specify a connection endpoint.

Capacity Management is an important feature within the SCA. Our example APCO-25 waveform requires C4FM modulation and AMBE Vocoding. Those capabilities could be delivered as waveform components, but that would compromise the waveform's independence. A more portable

solution would be to select (collaborate with if necessary) a Platform Component that provides the requisite functionality. The Platform Component could advertise the existence of that feature as an allocation property and then the Application Component could require the presence of that capability.

3.3 SCA MODIFICATIONS

3.3.1 Resource and Device Interface Refactoring

3.3.1.1 Overview

This release reworked the SCA 2.2.2 *Resource* and *Device* interfaces as a component of the other changes that occurred within the specification. Two primary changes occurred; the first of which decomposed the interfaces into more specialized, lower level interfaces; the second removed the *Resource*, *Device*, *LoadableDevice* and *ExecutableDevice* interfaces. The existence of the finer granularity interfaces provides the developer with the ability to create more secure, lighter weight components. The net impact of the changes is that the implemented components will support a set of operations and attributes roughly identical to those of the legacy interfaces, e.g. *Resource*, however they will require modifications to accommodate the new structure. The changes should be straightforward and minor in nature, e.g. changing the format of an operation invocation, and not require the introduction of new logic.

3.3.1.2 Resource Related Modifications

3.3.1.2.1 Resource interface changes

The new structure of the *Resource* interface supports the SCA optional composition pattern as well as the least privilege pattern employed within the JTNC APIs. The changes remove the specialized interface, *Resource*, and pass the responsibility of determining the inherited *Resource* interfaces that will be realized to the component. The flexibility of the approach becomes apparent when it is evaluated from the provider's perspective. Figure 37 highlights the *Resource* interface changes. The identifier attribute was moved to the *ComponentIdentifier* interface and the *start* and *stop* operations were moved to the *ControllableInterface* interface.

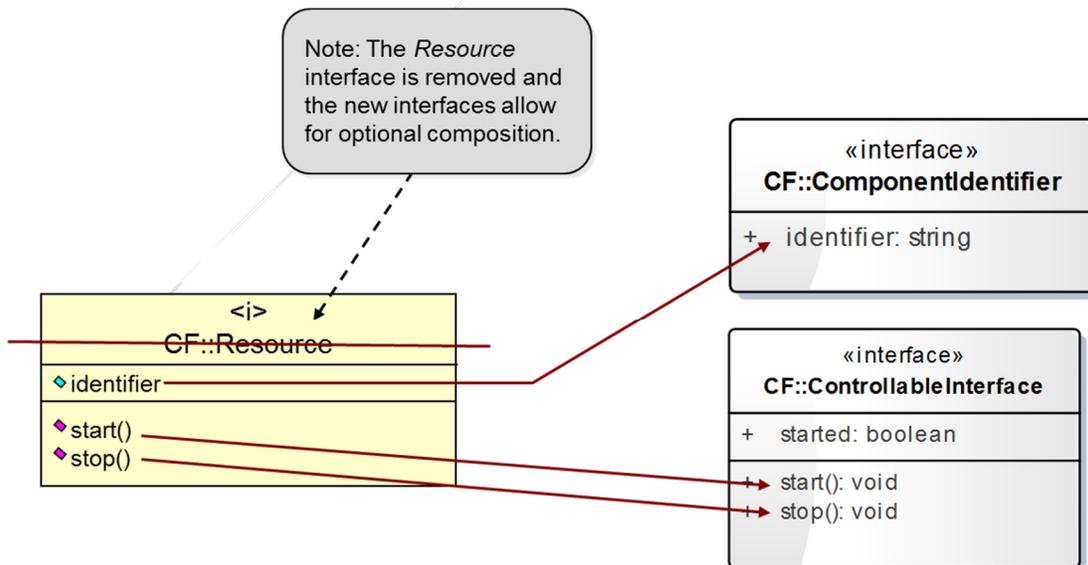


Figure 37 Resource Interface Refactoring

As seen in Figure 38, the equivalent, inherited *Resource* interfaces, may be realized optionally by an Application Component. Having the ability to realize these interfaces allows the component to be tailored to a product specific set of requirements. Eliminating unnecessary interfaces also increases the assurance level of the created component because the implementation will not contain any “dead” code and the finer granularity interface definitions allow the developer to expose only the interfaces and information that need to be provided.

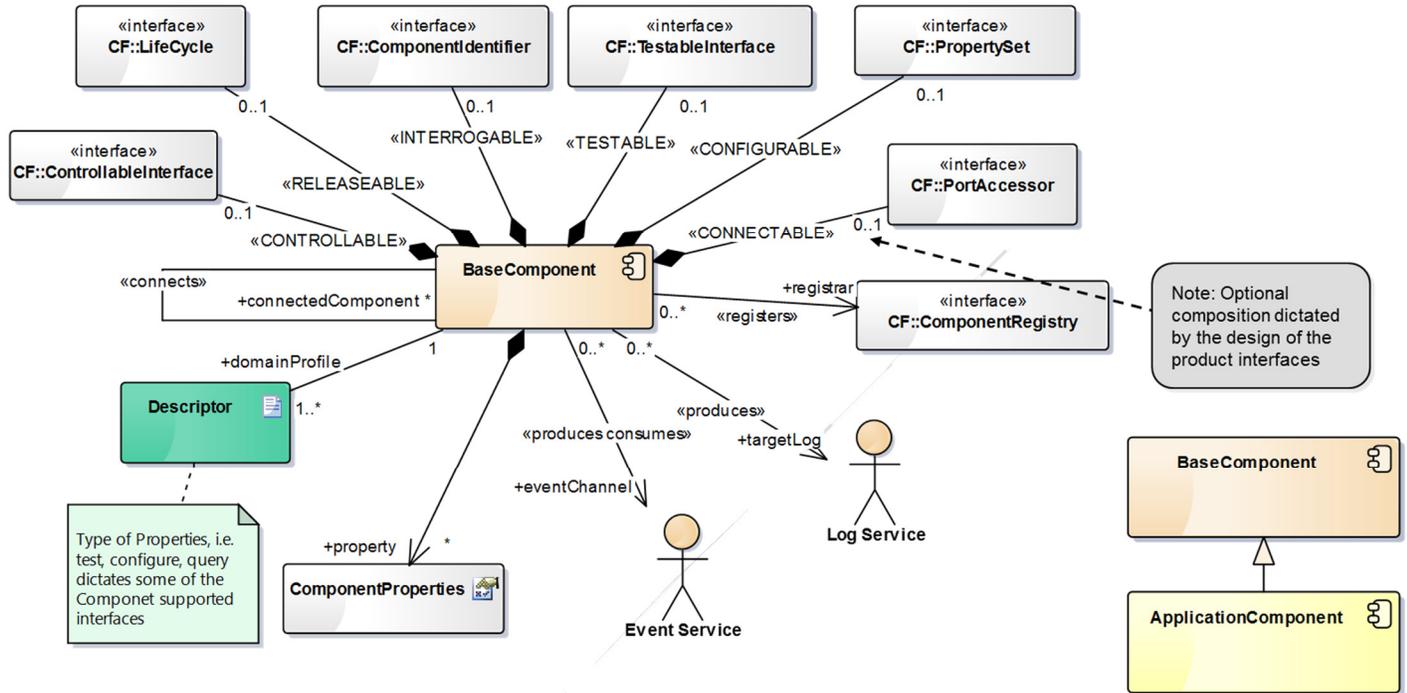


Figure 38 Application Component Optional Interfaces

3.3.1.2.2 ResourceFactory Interface Changes

The *ResourceFactory*, pictured in Figure 39, was also refactored. The *ResourceFactory* interface modifications take advantage of optional composition in a manner similar to that applied to the *Resource* interface, Figure 37, but it has two important distinctions. The *shutdown* and *releaseResource* operations were removed from the interface in lieu of an approach that aligns its life cycle management with the other CF interfaces, i.e. utilizing the *LifeCycle* interface. Secondly, the *ResourceFactory* interface was not removed, it was renamed to be *ComponentFactory*. The *ComponentFactory* interface was preserved because it retained the *createComponent* operation and it was renamed to reflect its new functionality that gives it the ability to create not only *ApplicationComponents* but also *PlatformComponents*.

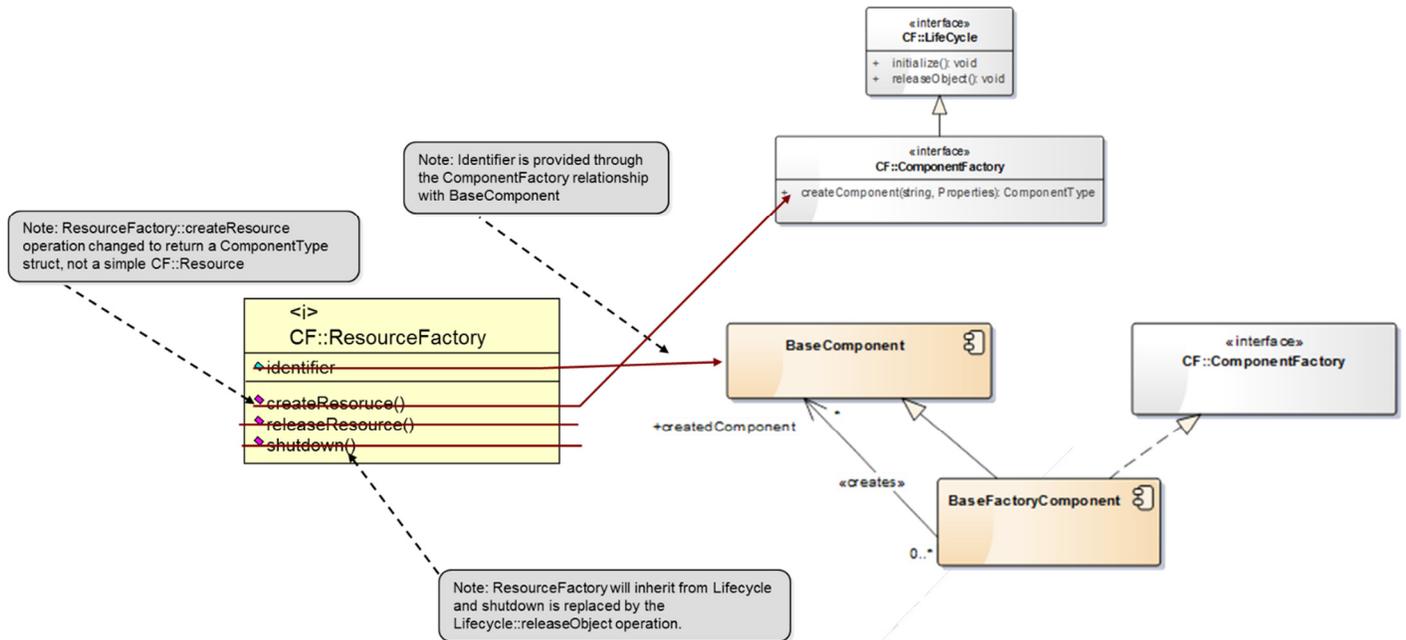


Figure 39 ResourceFactory Interface Refactoring

3.3.1.3 Device Related Modifications

3.3.1.3.1 Device and LoadableDevice interface changes

The *Device*, Figure 40, and *LoadableDevice*, Figure 42, interfaces were refactored such that they no longer have an inheritance relationship with one another or the *Resource* interface. The refactored components that provide the *Device* and *LoadableDevice* interface behavior utilize optional composition in a manner similar to the strategy used by the *Resource* interface replacement.

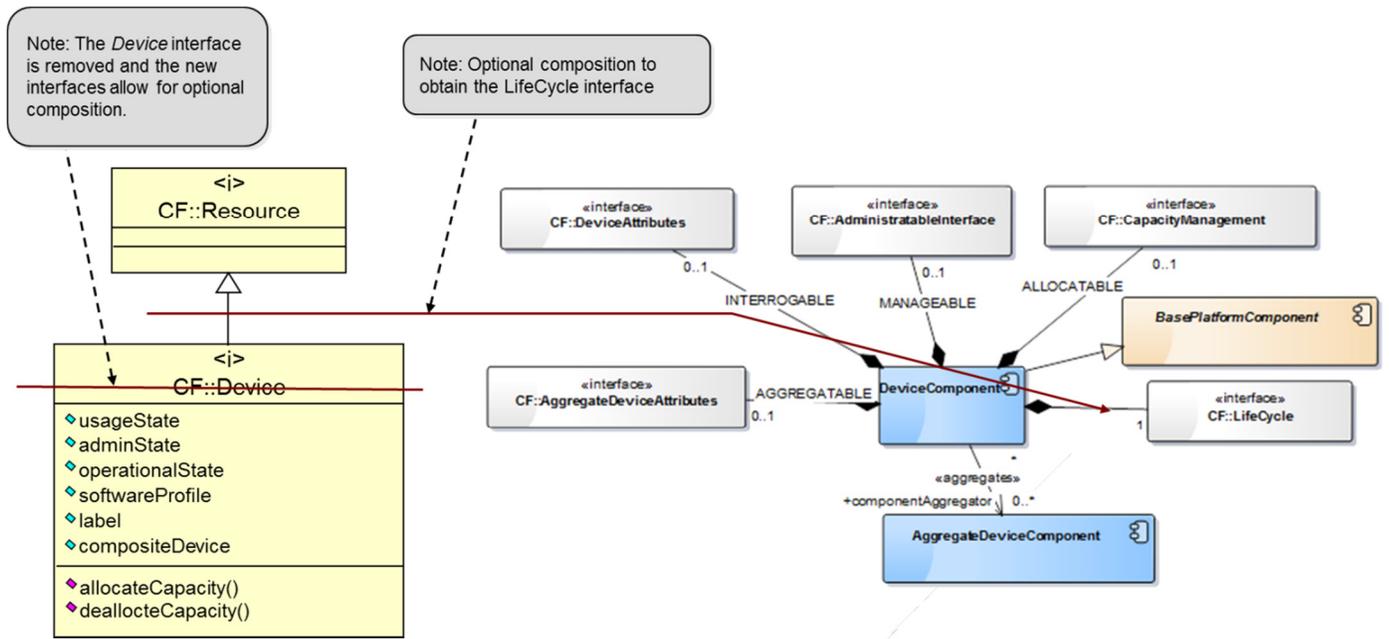


Figure 40 Device Interface Inheritance Refactoring

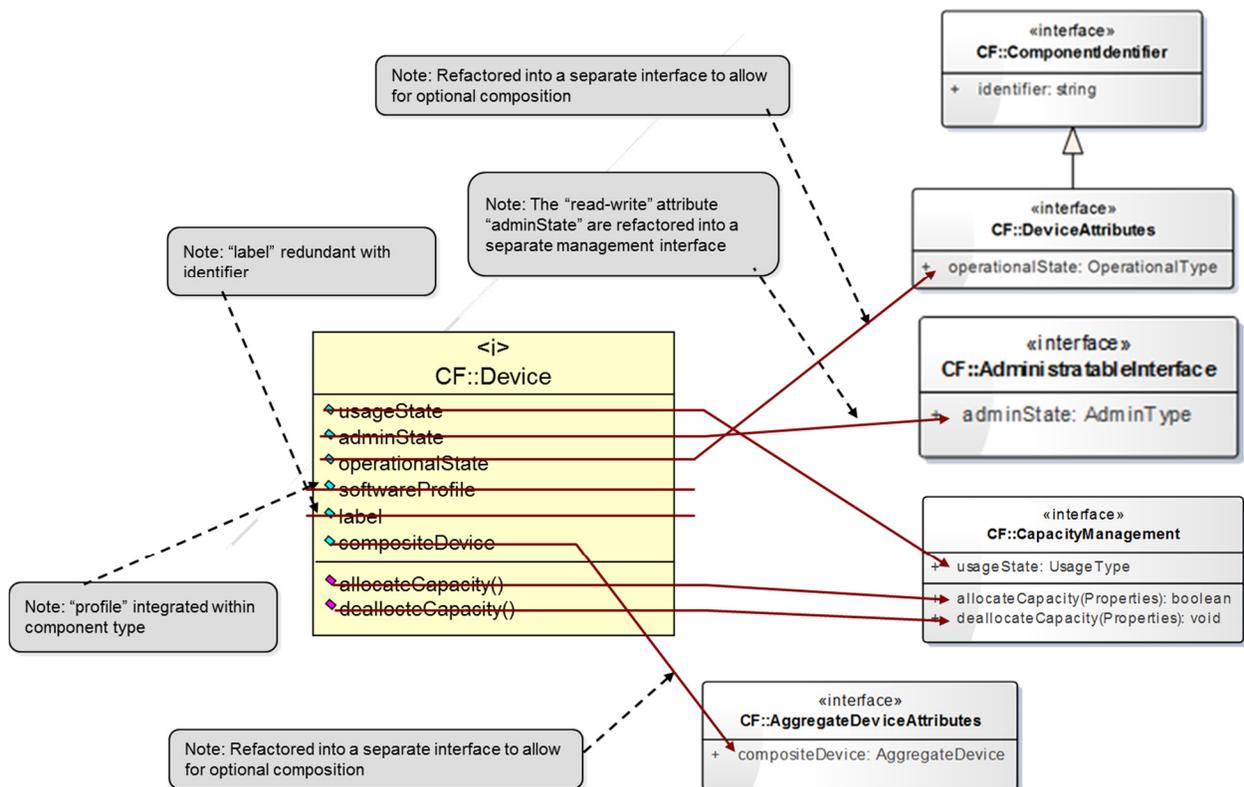


Figure 41 Device Interface Refactoring

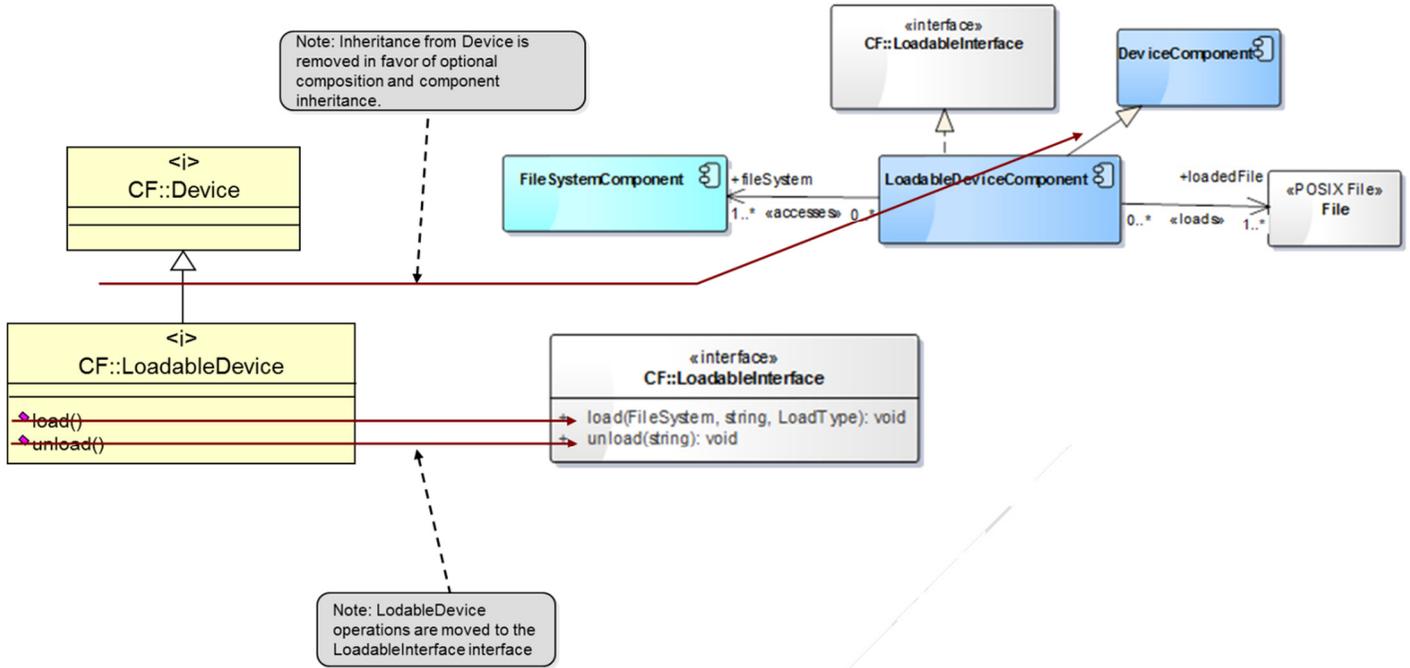


Figure 42 *LoadableDevice* Interface Refactoring

3.3.1.3.2 *ExecutableDevice* Interface Changes

The *ExecutableDevice* interface, Figure 43, was refactored so that it no longer has an inheritance relationship with the *LoadableDevice* interface. The *ExecutableDevice* operations were moved to a new interface, *ExecutableInterface*, which is accessed by the *ExecutableDevice* component via option composition. A new feature of the *ExecutableDeviceComponent* is that now it also can be optionally composed to provide loading functionality via the *LoadableInterface* interface.

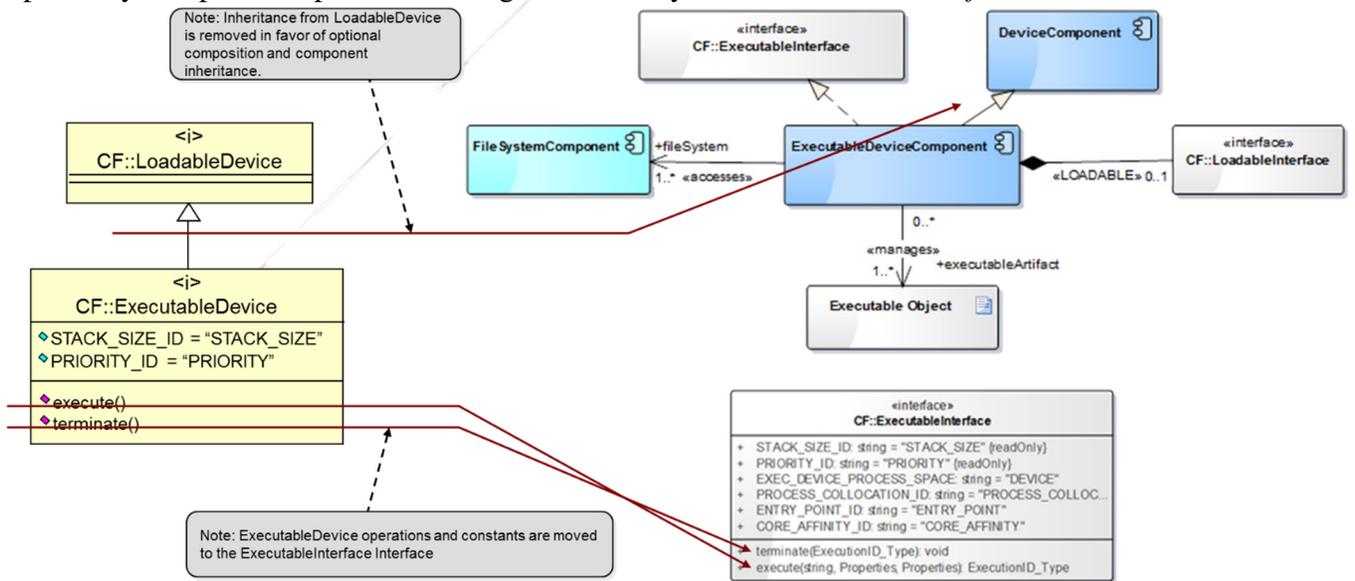


Figure 43 *ExecutableDevice* Interface Refactoring

3.3.1.4 Summary

The SCA 2.2.2 resource (i.e. application) and device interfaces were refactored to remove many of the operations and attributes from the top level interfaces and eliminate the inheritance relationship between those interfaces and the *CF::Resource* interface. The new interfaces are now accessible at the component level using the optional composition strategy. The rationale behind relocating the operations and attributes is to provide a developer with a mechanism to “right size” their components to align better with product requirements. Elimination of the inheritance relationship allows the components to circumvent the collocation prohibitions that are discussed in the Lightweight Components section 3.1.6.

3.3.2 Refactored CF Control and Registration Interfaces

3.3.2.1 Overview

SCA reworked the composition of the control and registration interfaces as part of the modifications that occurred within the specification. The most significant change was that the SCA 2.2.2 interfaces were refactored into smaller, more concise, standalone interfaces. The composition of these interfaces ensures that only the methods needed for management and registration of the “to be constructed” system are provided to consuming components. The presence of these prohibitions enhances the assurance profile of the platform because it follows the least privilege pattern, allowing only the necessary interfaces to be available and accessible. The refactoring also improves platform and system performance because it contains modifications that transform the SCA from a pull to a push model registration approach. Push model behavior is more efficient because it allows a component to pass along its information when it is ready and not wait to be called or encourage additional request, response cycles. The granularity of the information included within the pushes is also more efficient since the SCA approach now allows all of the component's information to be bundled within one push rather than forcing the components to invoke multiple requests for the same content.

3.3.2.2 *DeviceManager* Interface Changes

The *DeviceManager* registration operations, Figure 44, were collapsed and migrated from the interface. The migration was consistent with the principles of the least privilege pattern since it is unnecessary for a client that already has a reference to a *DeviceManagerComponent* to require an additional interface to obtain an endpoint to register itself. This move leverages the fact that the only components required to register with a *DeviceManagerComponent* are those that it launches, and it is reasonable to assume that the *DeviceManagerComponent* can provide its registration address as part of the launch parameters.

The registration process, which had been performed through the *DeviceManagerComponent*, *DomainManagerComponent* and *ApplicationFactoryComponent*, was refined as part of the redesign. SCA introduced a single capability, *ComponentRegistry* that could be associated with and used by any of those components. Component registration behavior was reworked to leverage a push model mode of operations which yields substantial performance improvements. Lastly, the implementation of the registries is much simpler because *ComponentRegistry* provides a general purpose registration capability that no longer needs to be tailored to register uniquely either service, device or application components.

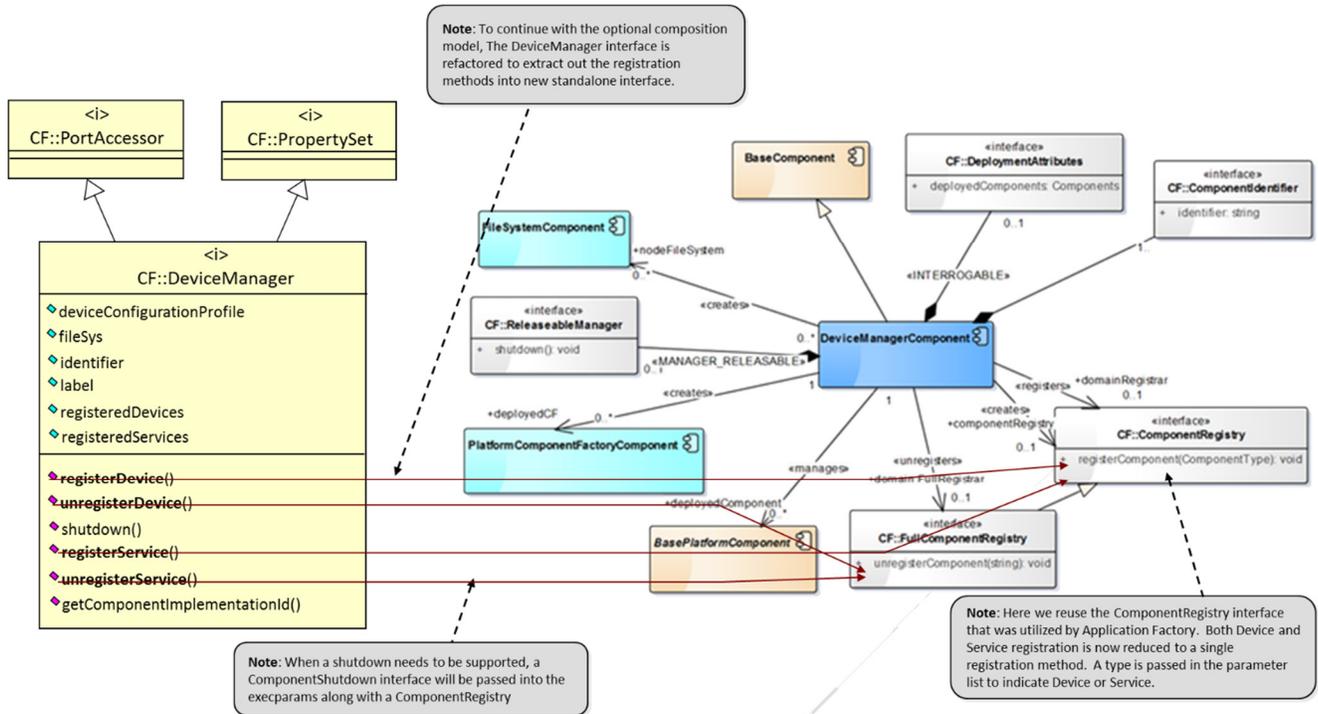


Figure 44 *DeviceManager* Interface Refactoring – registration operations

The refactoring removed the *DeviceManager* attributes from the top level interface. The predominant usage of these attributes before now was for interrogation by the *DomainManagerComponent* as part of pull model registration. These attributes are no longer needed within push model registration because the registering *DeviceManagerComponent* provides its corresponding values as part of registration. The refactored design provides an optional mechanism for the prior *DeviceManager* interface's attributes to be accessible, via the *DeviceManagerComponent*'s *ComponentType* data element, when an implementation finds it necessary to preserve the possibility of registered components being accessed externally.

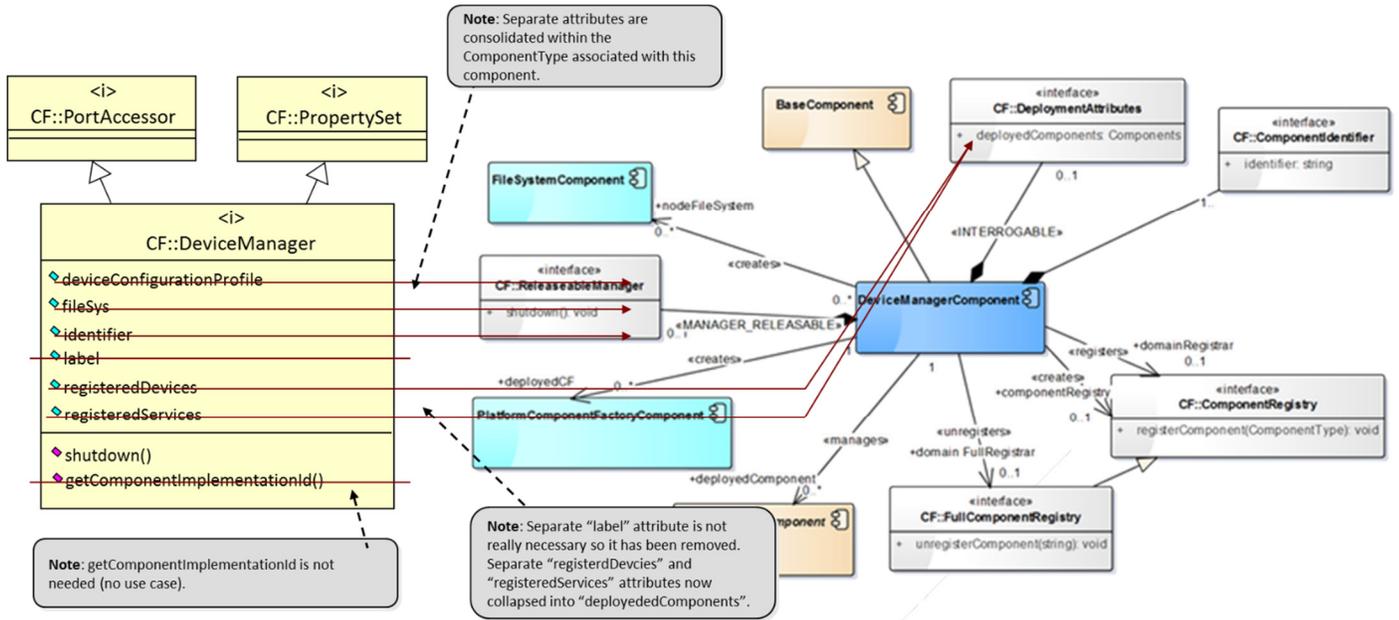


Figure 45 DeviceManager Interface Refactoring – attributes

The *DeviceManager* interface was removed and its inheritance relationship with the *PortAccessor* and *PropertySet* interfaces, Figure 46, was made optional per the optional composition pattern. The presence or absence of these interfaces is determined by the *DeviceManagerComponent*'s need for connections or implementation specific attributes.

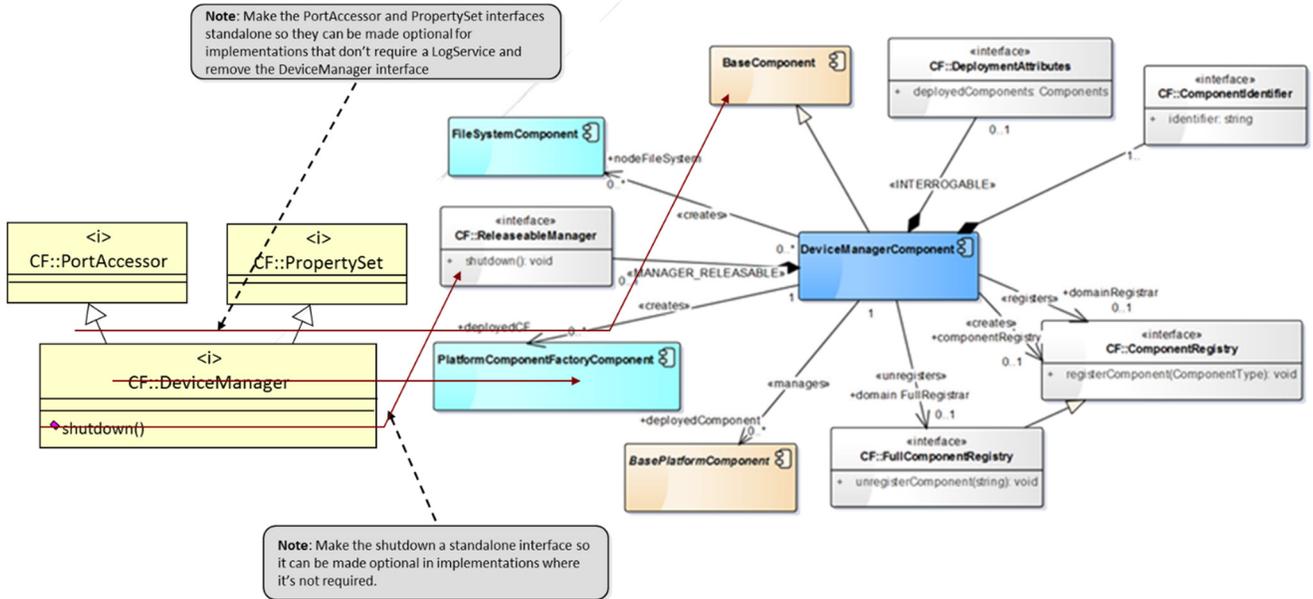


Figure 46 DeviceManager Interface Refactoring – miscellaneous operations

3.3.2.3 DomainManager interface changes

The *DomainManager* registration operations, Figure 47, were collapsed and migrated from its SCA 2.2.2 interface. The rationale behind the changes mirrors that provided for the corresponding changes in the *DeviceManager* interface. The *DomainManager* interface has an additional pair of registration related interfaces, used expressly for event registration, which were migrated to a new interface. Moving the event registration operations outside of the *DomainManager* interface aligns with the least privilege approach; however, the revised SCA did not integrate those services within the component registry. The event registration operations remained in a distinct interface because they have a wider range of potential users, from components launched by a *DeviceManagerComponent* to consumers that reside outside of the framework implementation, many of which should not have access to framework internals pertaining to registered components.

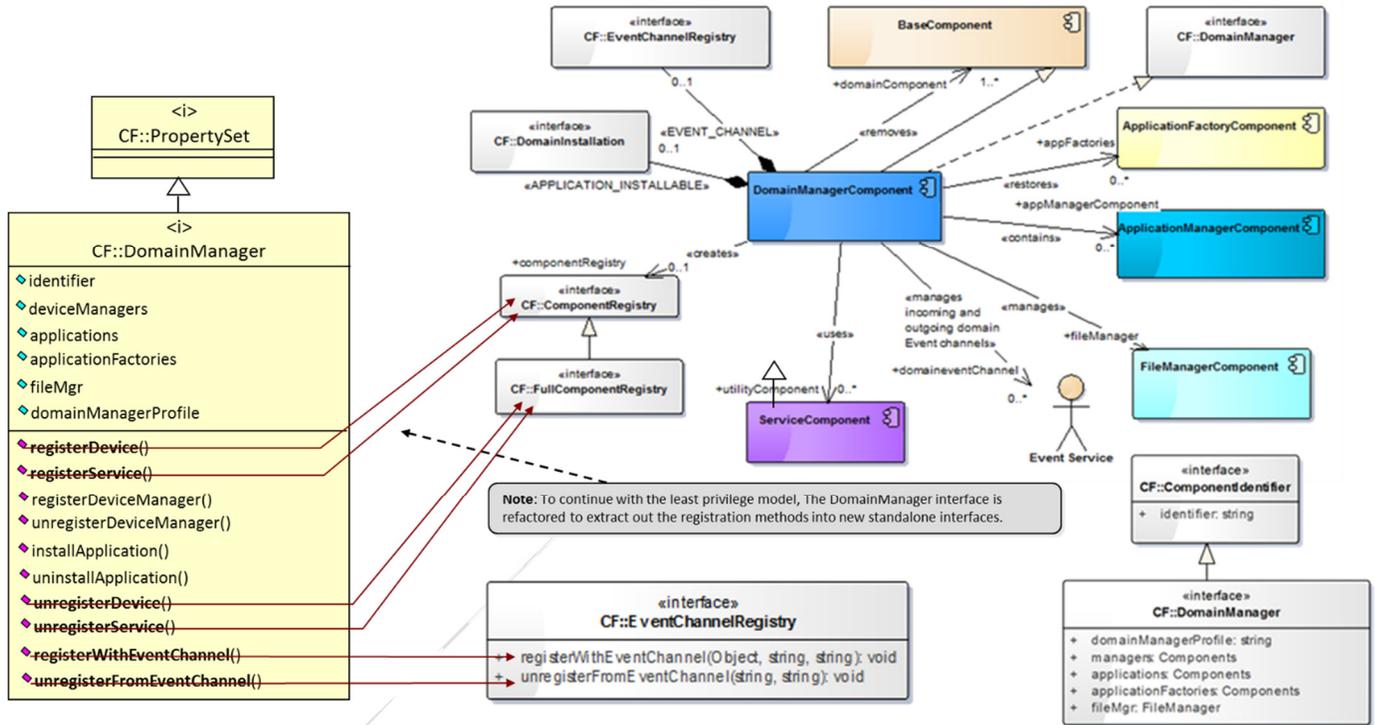


Figure 47 DomainManager Interface Refactoring – registration operations

The *DomainManagerComponent* no longer needs a separate manager registry since manager registration was integrated within the component registry. The application installation and uninstallation operations were also migrated away from the component. This migration was performed to satisfy scenarios, such as those associated with a static system configuration where no capability need exist to add or remove applications. Lastly, it should be noted that the *DomainManager* attributes were not removed from the interface. The attributes were retained because the *DomainManagerComponent* provides the interface between a platform domain and its external consumers, e.g. an external management system or user interface, and they provide the access point for those consumers to retrieve specific information regarding the system’s configuration.

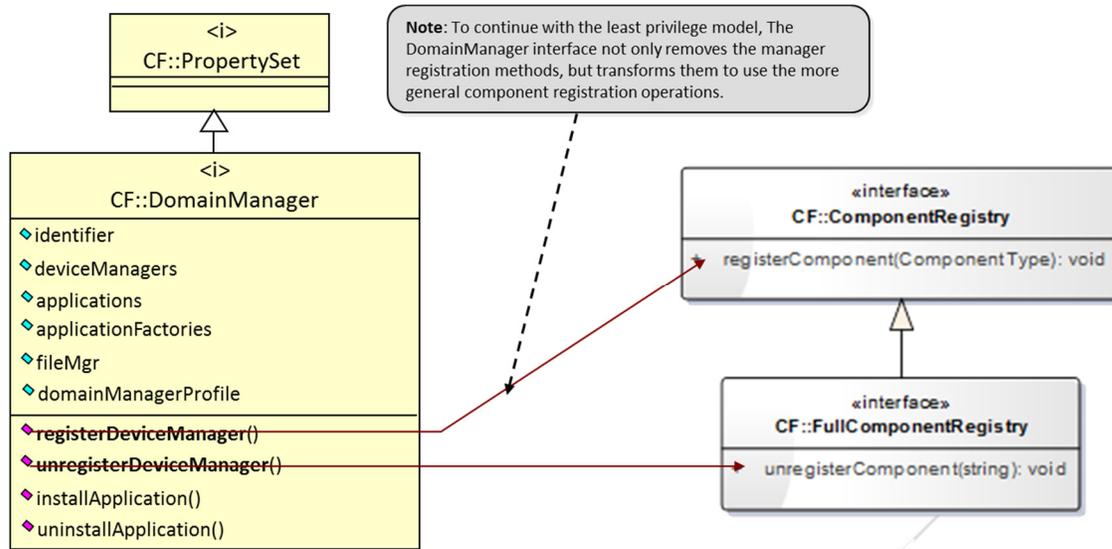


Figure 48 DomainManager Interface Refactoring – manager registration operations

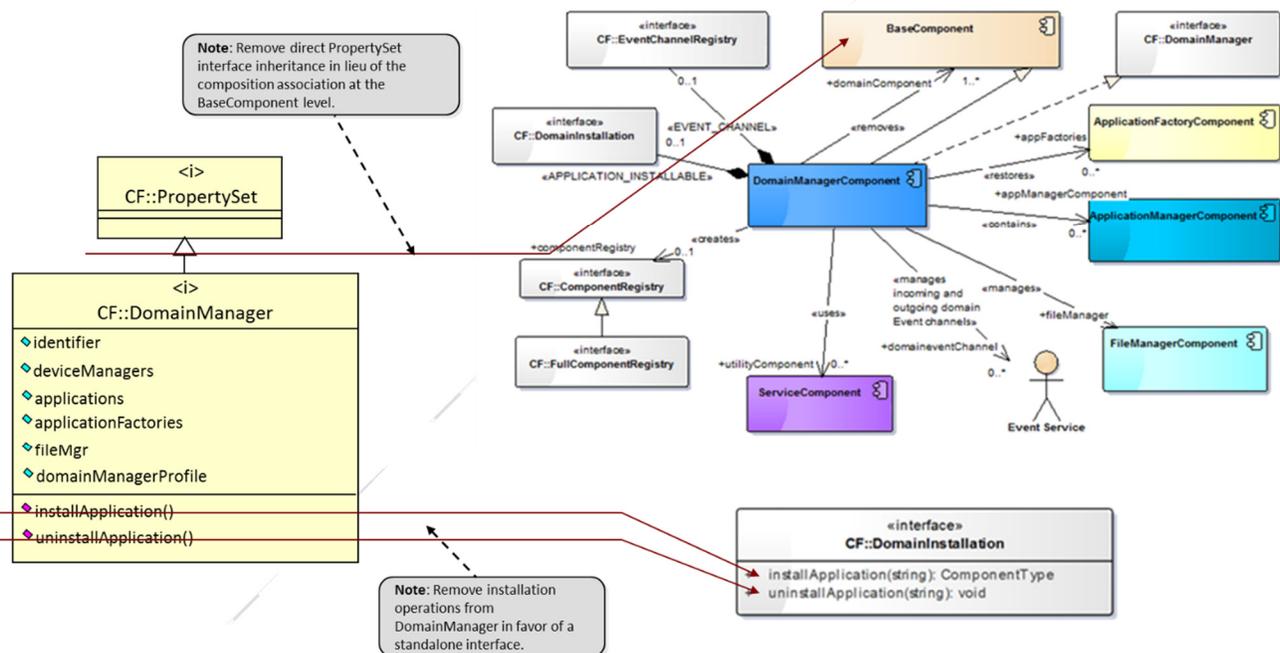


Figure 49 DomainManager Interface Refactoring – installation operations

3.3.2.4 Application Interface Changes

The *Application* interface, Figure 50, was refactored such that it removes direct visibility of many of the interface attributes. These attributes provide a way for clients to interrogate an application's run time internals. All of the information contained within the attributes is essential for proper framework operations, however several scenarios exist for which demonstrate that it is not needed by clients. Eliminating the interfaces improves system IA awareness and performance in accordance with the other push model enhancements. The interface was also renamed to

ApplicationManager to better align it with its role, providing the framework with a well-known point from which to manage the independently developed applications that are deployed within a domain.

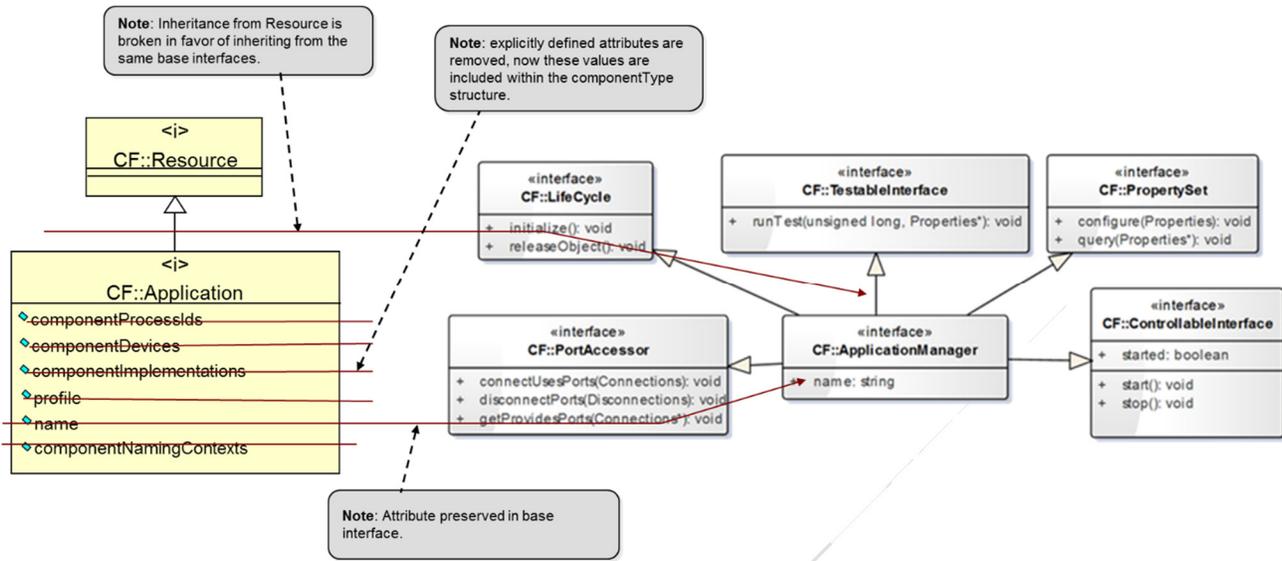


Figure 50 ApplicationManager Interface Refactoring

3.3.2.5 ApplicationFactory Interface Changes

This SCA revision provided a window of opportunity to clean up the *ApplicationFactory* interface, Figure 51. The *ApplicationFactory* interface is relatively simple so there were no large improvements to be achieved by introducing optional composition within the model. However, the interface had a redundant attribute, *identifier*, which was removed in order to clean up the interface specification and the contents of the *softwareProfile* attribute were moved within the *componentType* structure.

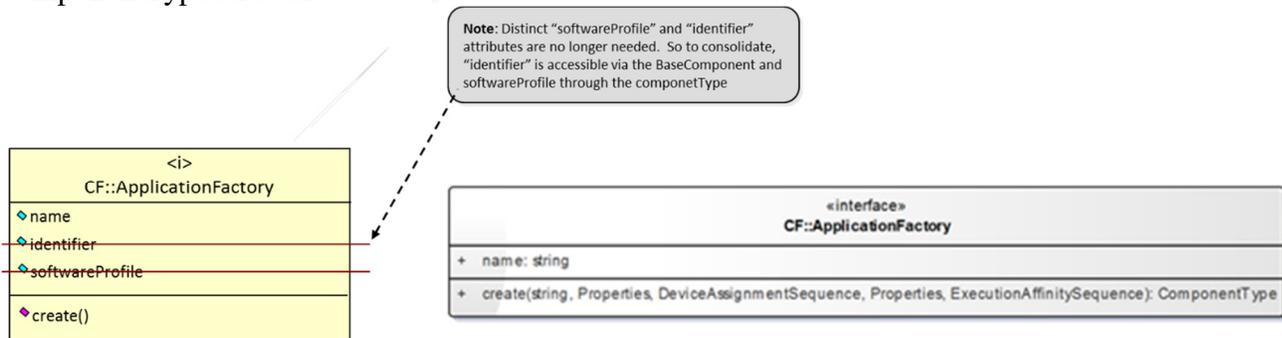


Figure 51 ApplicationFactory Interface Refactoring

3.3.2.6 Summary

The revised model of the SCA control and registration interfaces provides a set of access endpoints that allow a system developer to reduce the size and increase the assurance level of a core framework implementation. These modifications provide a standardized approach to lower product

development costs because there are fewer interfaces and requirements that need to be satisfied during the development process when unnecessary capabilities are omitted. The cost and size improvements are a welcome consequence of the revised approach, but the larger benefit is that SCA now allows a product development team to make intelligent determinations regarding their system's architecture and the information it will expose for external consumption.

3.4 WORKING IN AN SCA ENVIRONMENT

3.4.1 SCA 4.1 Development Responsibilities

3.4.1.1 Overview

SCA 4.1 contains several new component and interface definitions. An objective of the evolution of the specification was for it to provide clarifications that would help readers become proficient with the documents more quickly by better highlighting areas of interest. SCA 4.1 section 2.2 provides insight by identifying which developers are involved in realizing specific interfaces and components. Armed with that information a developer is better able to navigate through the sections of the specification that are a higher priority for their implementation.

3.4.1.2 Component Development Alignment

SCA 4.1 separates descriptions of the components hosted by the radio set from those provided by waveforms.

	Common Components		Base Application Components				Framework Control Components				Base Device Components		Framework Service Components								
	BaseComponent	BaseFactoryComponent	ManagedApplicationComponent	ApplicationControllerComponent	ApplicationComponent	ApplicationComponentFactoryComponent	AssemblyComponent	ApplicationFactoryComponent	DomainManagerComponent	DeviceManagerComponent	DeviceComponent	LoadableDeviceComponent	ExecutableDeviceComponent	AggregateDeviceComponent	FileComponent	FileSystemComponent	FileManagerComponent	BasePlatformComponent	PlatformComponentFactoryComponent	ServiceComponent	ManagedServiceComponent
Abstract Component	X	X																X			
Application Developers	X	X	X	X	X	X	X														
Device Developers	X	X									X	X	X	X				X	X		
Service Developers	X	X																X	X	X	X
Core Platform Developers	X							X	X	X	X				X	X	X				

Figure 52 General Allocation of Components to Radio Developers

SCA components are the elements that will be implemented by an SCA developer. Figure 52 allocates specific components of interest to the various communities of interest (and a designation for an Abstract Component) that provide products within a radio set architecture.

3.4.1.3 Component Products

The Abstract components encapsulate functionality that is not exposed directly to an external consumer or provider. Abstract components can be realized independently and used by multiple user facing components. BaseComponent is an example abstract component. It provides the core interfaces, relationships and requirements used by other SCA components. BaseComponent includes associations with the DomainProfile files and many of the fundamental SCA interfaces such as the *LifeCycle* interface. Application Developers, Device Developers, Service Developers and Core Platform Developers all create user facing components that have an inheritance relationship with BaseComponent, i.e. each of those components are responsible for providing interface realizations and fulfilling the applicable BaseComponent requirements.

Application Developers provide user facing, software intensive solutions such as waveforms that are deployed on the radio platform. In most cases a waveform will be delivered as a collection of Base Application Components. An application consists of an application controller(s), application components and (optionally) application component factories. The components are typically deployed separately and provide functionality, capabilities and associations as dictated by their operational requirements and those provided within the SCA model representations (which includes any levied by the Operating Environment such as the AEPs or their chosen Middleware). When the components are deployed separately, even the same component type can have differing configurations and constructs.

Device Developers provide software abstractions that mediate between system components and the physical hardware elements. Device Developers provide implementations of the Base Device Components. The components typically have a one to one relationship with a piece of system hardware and each one provides the functionality and capabilities dictated by the associations provided within the SCA model representations. Since Base Device Components need to work with a specific hardware element there are instances where they cannot be fully portable however it is advisable that Device Developers make every attempt possible to incorporate techniques and practices that promote portability.

Service Developers provide software abstractions that provide common functionality for multiple system components, be they applications, devices or other services. A service can be either a user facing product or a utility that provides additional capabilities to another system element. Services are unique within SCA because there are two distinct types of Framework Service Components, ServiceComponents and ManageableServiceComponents. ManageableServiceComponents should be used in scenarios where an SCA developer is providing the implementation. Since the developer is providing the design and implementation they are able to incorporate realizations of the SCA components and interfaces. ServiceComponents provide the abstraction for integrating capabilities, such as COTS components, that provide critical system functionality but do not contain source code that is accessible to the developer. In those cases, the service developer is limited to providing supplemental artifacts, such as domain profile files, that allow the service to be deployed by the framework.

Core Platform Developers provide software solutions that provide the essential Core Framework functionality, device and domain management and application creation and management, to a radio platform. Similar to device components, Framework Control Components are not explicitly

targeted to optimize portability, but by using the SCA constructs it is highly likely that they will be relatively portable, although they will contain localized areas that reference the radio set specific operating environment. Typically, Core Platform Developers will be responsible for the selection and/or integration of platform OE components. The SCA does not constrain the methods in which Framework Control Components interact with OE components; a difference from the way that application components interface with the OE, but an OE implementation is still governed by any applicable software security requirements. It is important to recognize that Framework Control Components may incorporate a wide array of extensions or enhancements, such as fault tolerant frameworks, as long as the mandatory capabilities are provided.

3.4.2 SCA Maintenance Process – How To Develop a New PSM?

3.4.2.1 Overview

Figure 53 depicts how a proposed SCA change is handled. Proposed changes could be anything from minor redlines to introducing a new capability within the specification. Successfully implementing changes is a collaborative process that involves the change submitter, the Interface Control Working Group (ICWG) staff, the ICWG working panel and the JTNC. In summary, once an SCA enhancement is submitted, the working panel will collaborate with the submitter to determine if or how the enhancement should be integrated within the specification. Once the final revisions are complete, the ICWG staff will work with the JTNC to develop a strategy regarding when and how the change will be released. Detailed descriptions of the individual process actions are beyond the scope of this document but may be obtained by contacting the ICWG staff at jtrs-sca@spawar.navy.mil.

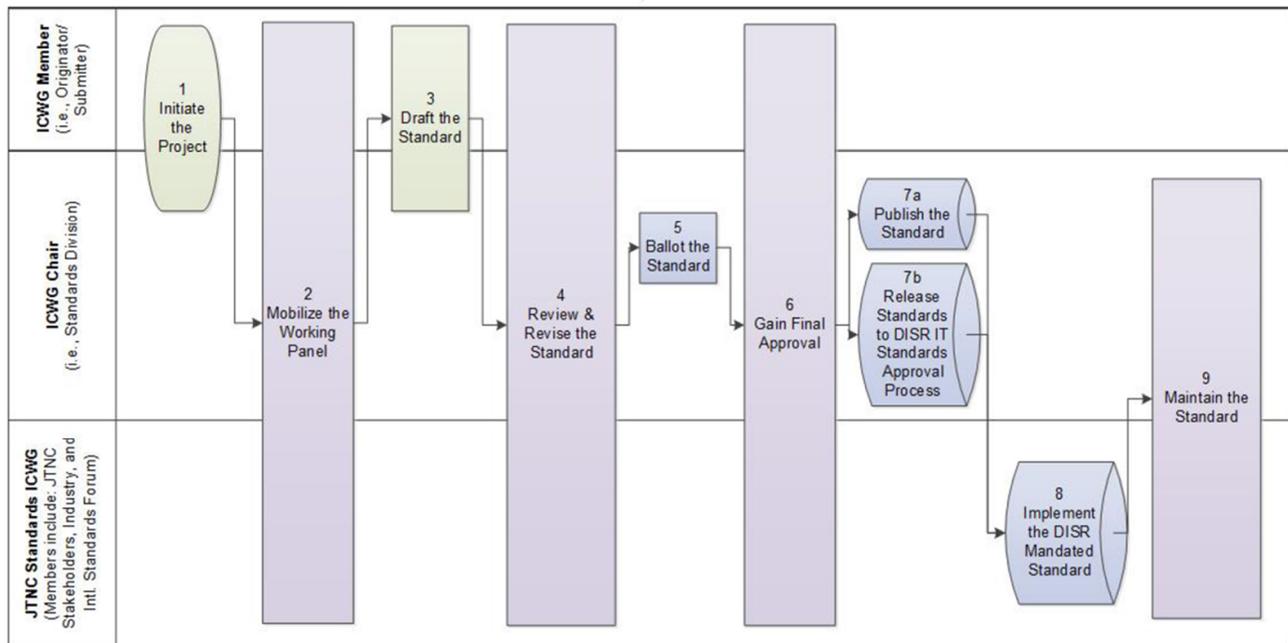


Figure 53 SCA Change Proposal Process

3.4.2.2 SCA Change Proposal Process – Submitter Roles and Responsibilities

SCA has evolved largely based upon inputs, new ideas and lessons learned, from its community of users. Consequently, inputs from submitters are an essential part of the process. The primary role of the submitter is to collaborate with the ICWG staff and working panel to communicate the reason for or rationale behind a change. The submitter will provide the information via a change proposal form, discussions or documentation. Any information not provided as part of the submission will be obtained via requests initiated by the working group.

SCA exercises the defined process with a focus on extending the content of the specification as directed by the user community's needs and requirements. The SCA 4.1 work began with a PSM definition, equivalent to that of SCA 2.2.2, a vision of how the specification should evolve and an outline of an additional PSM. The initial working panel neither had the available staff to define an additional PSM nor the desire to expend a large amount of effort working on a PSM that would not be used. Therefore, the group decided to proceed with a "need based strategy" that would wait for a community of interested users to drive the expansion of additional models.

Using the needs based strategy; a submitter would develop an idea for a new PSM. The proposal, step 1 within Figure 53, could be an errata statement, a document that appears ready for inclusion within appendix E or anything in between. The working panel will work with the submitter to refine the proposal so that it will be ready for presentation to the full ICWG in step 4. Beyond that point the idea will be fleshed out and refined until it reaches a point where it can be approved in step 5. Step 5 represents a decision point where the change will be balloted, but practically speaking it is unlikely that a full version of a new PSM proposal will reach this point if it doesn't have majority support of the working panel or ICWG voting members.

A proposal for a new PSM submission should be developed in a format equivalent to that of the existing appendices. It should include information equivalent to those in the current specs, e.g. if an XML schema version of the descriptor files were proposed, it should support the capabilities of the Document Type Definition (DTD) based descriptors. If the new proposal omits some of the preexisting constructs then those omissions should be interpreted as a prompt to revisit those elements to see if they should be removed from there as well.

If a submitter were to propose a new transport mechanism, that proposal should strive to present a solution based on Standard technologies which excludes features detrimental to common wireless communication device attributes such as performance, sizing or security.

3.4.3 SCA Naming Conventions

The structure and appearance of the new SCA is vastly different from that of SCA 2.2.2. In addition to the reorganization, the revisions introduced several new elements within the SCA lexicon.

The introduction of the component model created a collection of new elements that described the SCA components. SCA 2.2.2 and earlier versions leveraged the component concept, but used text formatting conventions to distinguish whether or not the interface or the component was being referenced.

Multiple new SCA features: decomposing the IDL into smaller segments, more widely applying the principle of least privilege, introducing optional composition within the core framework had a side effect of creating new interfaces within the specification.

Therefore, the SCA designers perceived this as the best opportunity to overhaul the names of the existing SCA constructs if it ever was going to be undertaken and felt that this would be a good

opportunity to introduce a set of conventions that could influence the naming of any future constructs.

3.4.3.1 Component Naming Conventions

- “Component” should be at the end of the name to indicate that it is a component. For example, use “BaseComponent” instead of “ComponentBase”.
- When a component is directly associated with an interface, e.g. *CF::ApplicationManager*, the interface name is the beginning of the component name so this example becomes an *ApplicationManagerComponent*.
- Use “Base” as a prefix for the name of a conceptual/generic/abstract component, e.g. *BaseComponent*.
- Use a descriptive noun which describes the role of the component, e.g. *DeviceManager*, when a component does not have a corresponding interface.

SCA 4.1 Name
BaseComponent
BaseFactoryComponent
ComponentManagerComponent
ManageableApplicationComponent
ApplicationControllerComponent
ApplicationComponent
ApplicationComponentFactoryComponent
AssemblyComponent
ApplicationFactoryComponent
ApplicationManagerComponent
DomainManagerComponent
DeviceManagerComponent
DeviceComponent
LoadableDeviceComponent
ExecutableDeviceComponent
AggregateDeviceComponent
FileComponent
FileSystemComponent
FileManagerComponent
BasePlatformComponent
PlatformComponentFactoryComponent
ServiceComponent
ManageableServiceComponent

Figure 54 SCA Components

3.4.3.2 Interface Naming Conventions

- Use nouns for interface names – component level interface names should describe the role of the interface, the prefix of sub-component level interfaces should be an adjective which describes the function provided by the interface.

- Do not include words that conflict with the function of an interface, e.g. “object” or “component” within the name – therefore *TestableObject* was switched to *TestableInterface*.
- “Interface” should be at the end of the name to indicate that it is a sub-component level interface. For example, use *ControllableInterface* as a name for the interface that is used by the *ApplicationManagerComponent*.
- Use the first operation defined in the interface as the prefix of the name of a subcomponent interface.

SCA 4.1 Name
ComponentFactory
ComponentIdentifier
PortAccessor
LifeCycle
TestableInterface
PropertySet
ControllableInterface
ApplicationManager
DeploymentAttributes
ApplicationFactory
DomainManager
DomainInstallation
ComponentRegistry
FullComponentRegistry
EventChannelRegistry
ReleasableManager
AdministratableInterface
CapacityManagement
DeviceAttributes
AggregateDeviceAttributes
LoadableInterface
ExecutableInterface
AggregateDevice
File
FileSystem
FileManager

Figure 55 SCA Interfaces

The interface naming conventions were applied as guidelines and not mandates for the preexisting interfaces. In most cases, interface names were only changed if they were deemed to have a profound impact on the readability of the specification and they would not result in a disproportionate impact on the preexisting SCA compliant code base.

3.5 SCA Q&A

3.5.1 What elements of OMG IDL are allowed in the PIM?

3.5.1.1 Overview

The SCA Platform Independent Model (PIM) is communicated two ways within the SCA. The PIM is communicated via the UML models that are documented within the specification and accompany the document. Per Section 3, the elements of the PIM are also communicated in IDL; “OMG IDL is the standard representation for the standalone interface definitions within the SCA platform independent model”.

The IDL representation of the “SCA PIM” is a fixed entity that has its composition determined by the entity that developed the specification. Consequently, the question posed in this section is irrelevant because there is no latitude for an SCA user to consider adding additional elements to the formal “SCA PIM”.

3.5.1.2 PIM Background

The OMG defines a PIM as a representation that exhibits a degree of platform independence so as to be suitable for use with a number of different platforms of similar type. They suggest a common technique to employ in order to achieve platform independence is to target a system model for a technology-neutral virtual machine.

3.5.1.3 PIM usage for SCA developers

Within a model driven architecture approach many transformations can occur within a single abstraction layer. Therefore a user of the SCA PIM might choose to introduce several layers of refinement of the SCA constructs as part of the system design and development process while maintaining a platform independent model. The question of what IDL elements should be used is very relevant for developers who are planning on refining their PIMs. If a waveform is intended to be portable across multiple connection-mechanisms, then its IDL PIM should not introduce any elements beyond those specified in Appendix E- (reference [8]).

3.5.1.4 Future PIM evolution

The projected evolution approach for the SCA PIM is that it will migrate to a model which relies exclusively on UML. In that scenario the PIM would be fully integrated within a tool-based, largely automated software development process. System developers within this approach would execute all of their PIM refinement in the tool and in UML. When the modeler was ready to transition to a platform specific representation, this approach would treat IDL as a platform specific realization and the tool would facilitate the mapping to the target technology. Unfortunately we are not yet at a point where we can utilize this approach because the state of the art tools do not sufficiently support an automated generation of our desired mappings.

Nonetheless, in this scenario, the PIM would still be governed by the constructs defined in Appendix E (reference [8]); however the restrictions would be less apparent to the system architect.

3.5.2 What is the Impact of the SCA Port changes?

3.5.2.1 Overview

One of the SCA changes that has drawn considerable interest has been the refactoring of the port related interfaces. The specification introduced a new interface, *PortAccessor*, which consolidates the *Port* and *PortSupplier* interfaces. The new interface represents a change in the means in which an application or port user interacts with other framework elements or users. However the modification affords the SCA with several optimization opportunities and there are techniques that can be used to minimize the impact of the changes.

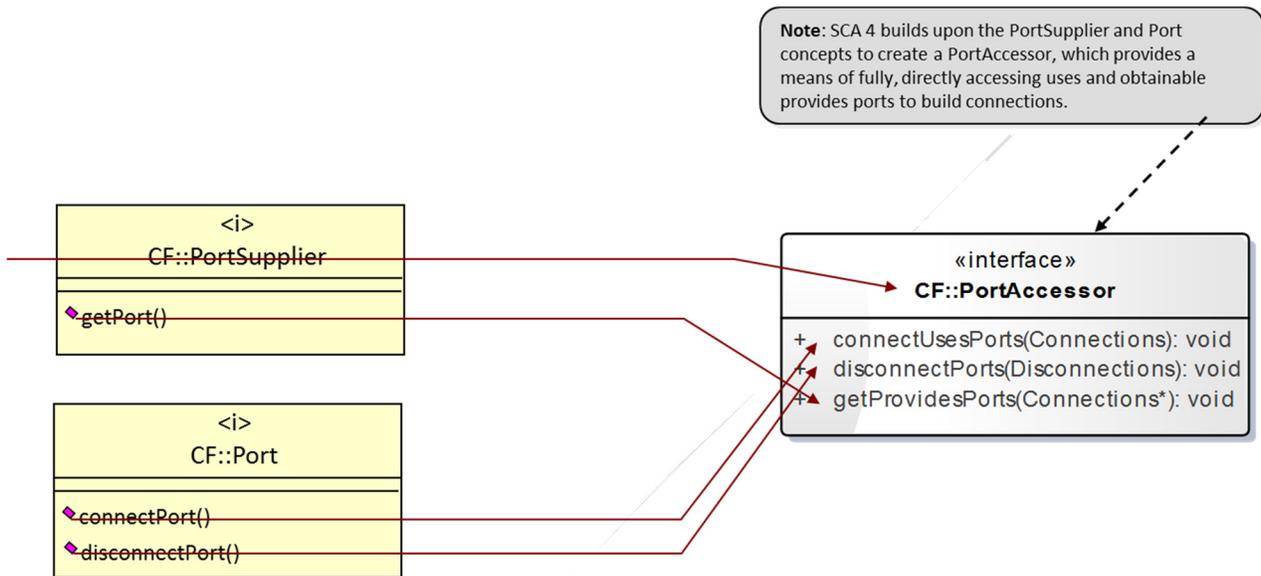


Figure 56 Port Interface Refactoring

3.5.2.2 Port Revisions

The *PortAccessor*, interface has three primary distinctions from the earlier SCA configuration, the interface contains information for both port providers and users, the consolidated port behavior is now integrated with the parent interface through an inheritance relationship (the earlier *Port* interface did not have a defined relationship) and the cardinality of the operations has been changed to accommodate multiple ports on one invocation.

Consolidating the ports into a single inherited interface eliminates the need for a separate uses port servant because the behavior associated with the client is now integrated within the interface realization on the uses side component. Collectively, the changes provide a performance enhancement because during the formation of connections there is no longer a need to obtain distinct uses ports because they are part of the component. The revised cardinality on the operations provide a means to reduce the number of required operation calls during the connection establishment process because many connections can be made with a single call.

The *PortAccessor* modifications also pave the way for enhanced connection management functionality. Integrating the port functionality within the provides side of the interface adds a release capability on that side. The introduction of which allows a provides port to have full

lifecycle support associated with a connection, the implication being that a connection could be created and destroyed on the provides side, so dynamic port management could occur.

3.5.2.3 Interface and Implementation Differences

The following changes exist on the uses port side:

- the implementation no longer has to create an association with the *Port* interface,
- the client will need to change any of its *Port* references to *PortAccessor*,
- the realized operation names will change from *connectPort* and *disconnectPort* to *connectUsesPorts* and *disconnectPorts*.

The logic change associated with the operation change should be straightforward as it will only need to be amended to accept lists of connection endpoints rather than a single endpoint.

A comparable set of changes will need to be performed on the provides ports:

- the interface definitions will change, which in turn will force an IDL recompilation
- the realized operation name will change from *getPorts* to *getProvidesPorts*

As a component of these changes, the new operation will return a void rather than an object reference and the parameter will no longer be a name, but a connection structure.

3.5.2.4 Implementation Implications

There are steps that can be employed to minimize the impact of the port related changes on an implementation. Figure 57 highlights some of the similarities and differences of the SCA 4.1 and SCA 2.2.2 port and connection implementations.

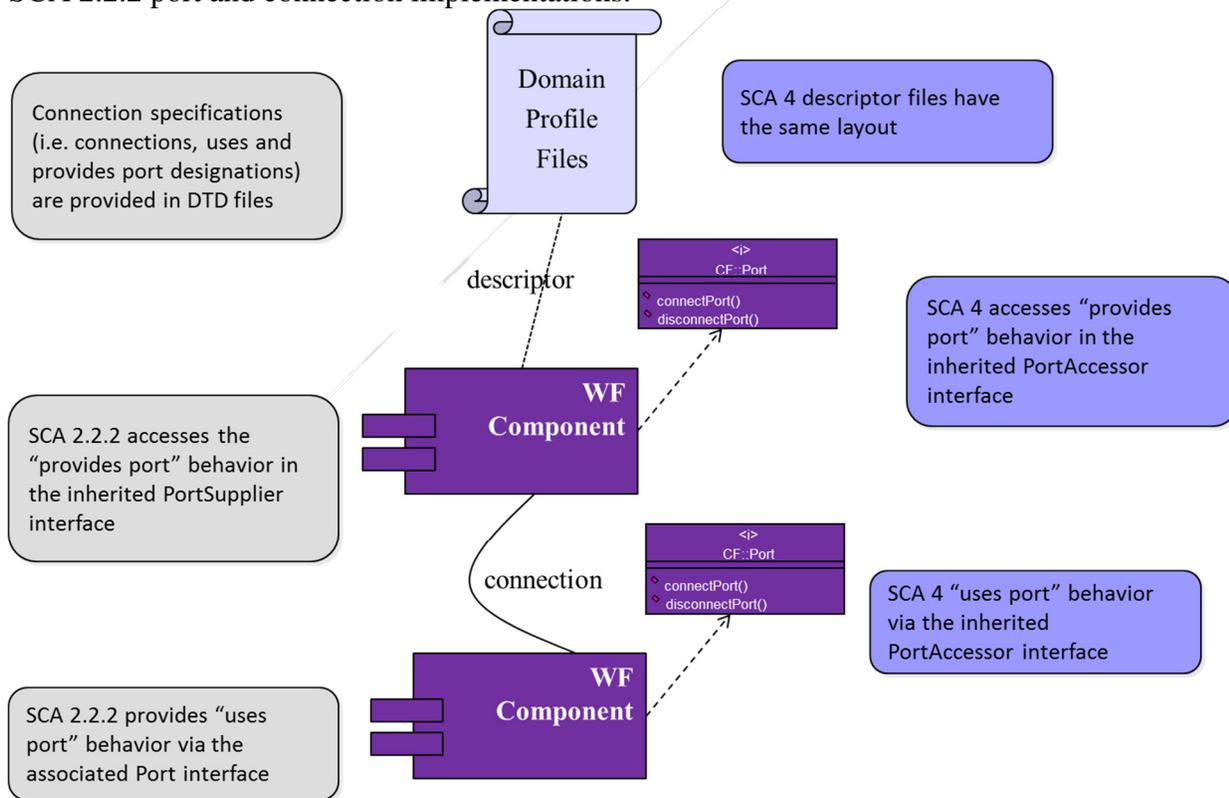


Figure 57 Port Implementation Differences

An SCA implementation could choose to create a “new” realization of the *PortAccessor* interface. This would be a reasonable approach to take, especially in instances where there are a limited number of locations where the code would need to be redone. This approach would likely be palatable in these situations because, in an unenhanced implementation the *PortAccessor* operations should not have very complex application logic.

There are a number of other scenarios where there may be more motivation to preserve the existing *Port* and *PortSupplier* implementations and to maximize the backwards compatibility of the SCA 4.1 design. A new *PortAccessor* realization can be introduced as a façade for the *PortSupplier* and *Port* realizations. In that role, the responsibility of the *PortAccessor* would be minimal; it would be responsible for managing the distinctions between the operation signature differences. Secondly, the developer can take advantage of the fact that many of the new features are optional. Therefore the differences between the SCA 2.2.2 and 4.1 implementations could be minimized by modeling the implementation using obtainable ports and not taking advantage of the “port aggregation” feature, thus minimizing the need to perform extensive code modifications. Lastly, in an approach that is similar to the façade pattern, the code could retain the *Port* interface and perform realization in a language specific PSM. A component and its underlying *PortAccessor* realization would have a delegation relationship or association to the *Port* PSM.

3.5.3 Rationale for DeviceManagerComponent Registration

Requirement SCA216 specifies that upon start up a *DeviceManagerComponent* has the responsibility of registering with a *DomainManagerComponent*.

A *DomainManagerComponent* is used for the control and configuration of the system domain. While not part of the original SCA objectives it is the case that in many instances a *DomainManagerComponent* can be viewed as platform agnostic and implemented in a fairly portable manner.

A *DeviceManagerComponent* manages a collection of *BasePlatformComponents* which are targeted for a specific node. A *DeviceManagerComponent* can be written using a fairly portable approach or it could be developed in a target specific manner in conjunction with the *BasePlatformComponents* that it will be hosting or its target Operating Environment.

Regardless of the selected development approach, the presence of requirement SCA216 allows for decoupled, either by provider or philosophy, implementations of the two components. This requirement provides a foundation that guarantees that even if the components are developed independently, they can be integrated at runtime via the *DeviceManagerComponent* registering with the domain via the *DomainManagerComponent*'s associated *ComponentRegistry* reference.

3.5.4 Rationale for Removal of Application Release Requirement

Earlier SCA versions contained a requirement, equivalent to the following statement: "The *ApplicationManager::releaseObject* operation for an application should disconnect ports first, then release its components, call the *terminate* operation, and lastly call the *unload* operation on the *DeviceComponents*."

SCA contains the following sequence diagram that demonstrates one scenario describing the steps associated with an application's release.

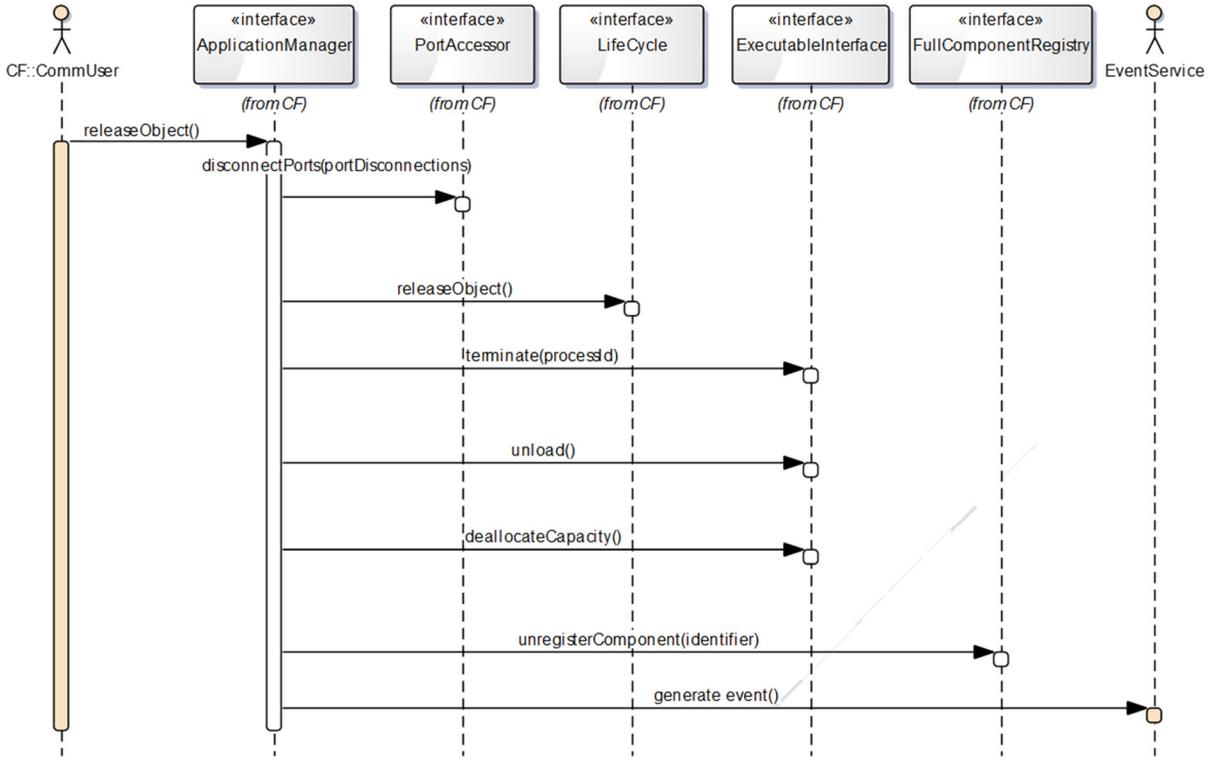


Figure 58 Sequence Diagram depicting application release behavior

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

The consensus was that this requirement was no longer necessary because the well-defined ordering specified within the requirement did not need to be preserved because the *ApplicationManager* interface contains individual requirements for disconnect, terminate, release and unload behavior and the relative ordering of those calls is dictated by their semantics.

3.5.5 Removal of the UML to Language Mappings

The platform independent philosophy of the SCA lends itself to a countless number of platform specific representations. The platforms could be specific with respect to processor architecture, middleware, programming language or other attributes. Earlier SCA versions paved the way for some of the PIM to PSM transitions to occur by defining a set of UML to programming language mappings. An ICWG working panel initiated this work to support the definition of a set of SCA specific, optimized mappings.

Upon reinvestigation of this decision it became apparent that it ran counter to a number of the other SCA related objectives related to the use of open architectures and industry Standards. SCA has decided to use IDL as a platform independent representation and undertaking the responsibility of defining a new set of mapping rules did not seem wise if we were not going to use them immediately and the fact that there are other groups, such as the OMG UML committee, that would be better qualified to define the mapping rules and an associated set of compliant tools.

Ultimately, the existence of a set of UML to language mappings and tools would better serve some of the SCA's long term, model based development objectives, but the supporting infrastructure does not currently exist to initiate a transition to this approach.

3.6 FUTURE ENHANCEMENTS

3.6.1 Component Life Cycle

3.6.1.1 Overview

SCA provides full lifecycle support for some of the Core Framework Control components, e.g. what happens when DeviceManagerComponents transition into and out of existence, but there is a lack of concrete guidance regarding the lifecycle of BaseComponent based components. A fully fleshed out lifecycle for these components would include a set of appropriate states and a description of the transitions that exist as the components, in particular ApplicationComponents, are installed or managed and a description of what environmental preconditions are required to bring a radio platform into existence.

3.6.1.2 BaseComponent State Model <Requesting Additional Input>

This instance of the BaseComponent state model semantics (legitimate operations and transitions) depends on the presence of the *LifeCycle* interface and support of the CONTROLLABLE flag.

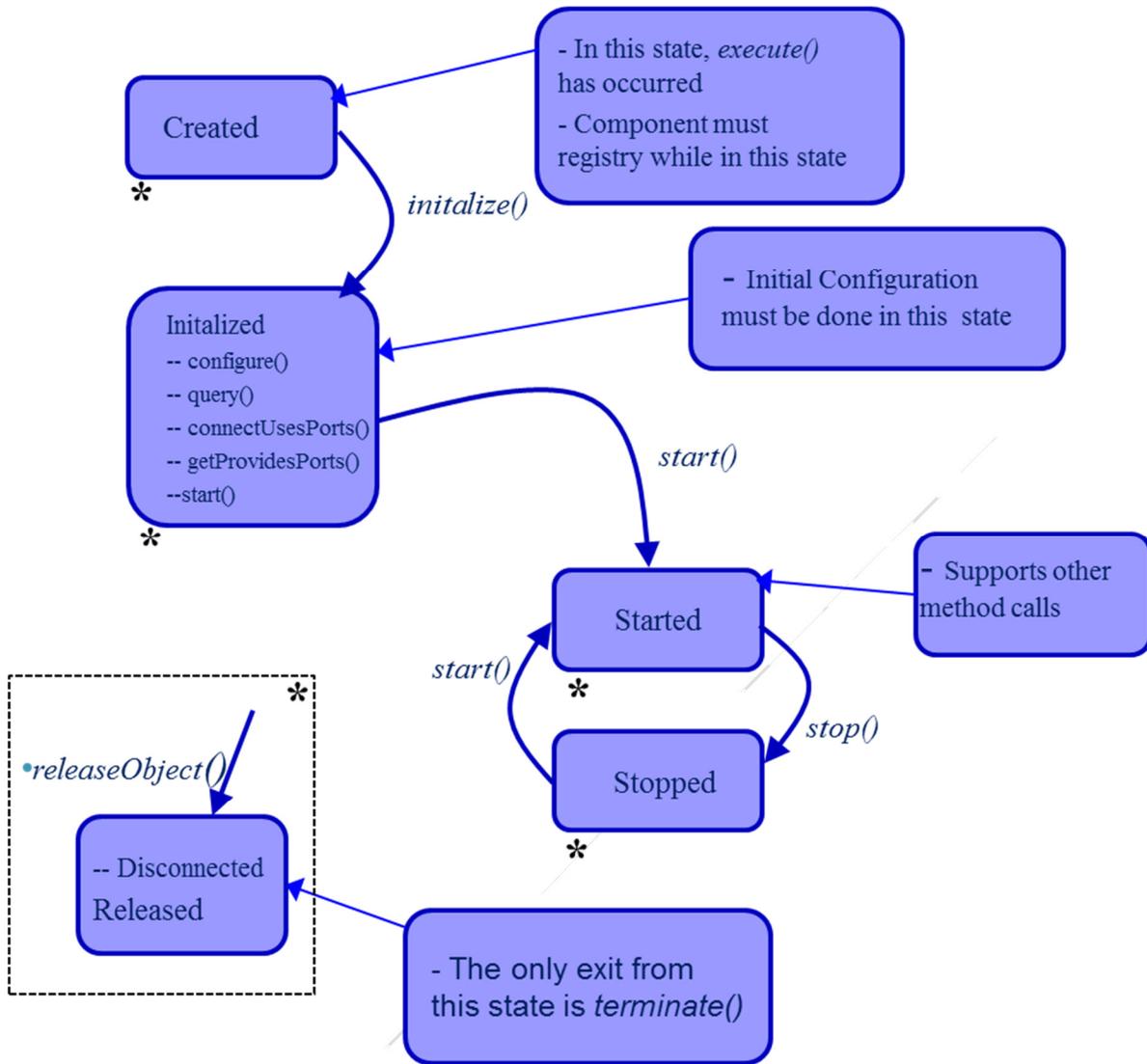


Figure 59 Component Life Cycle

[Note: Soliciting community for additional content to be added here. Please submit input to jtrs-sca@spawar.navy.mil.]

4 ACRONYMS

Abbreviation	Definition
AEP	Application Environment Profile
API	Application Program Interface
CF	Core Framework
CORBA	Common Object Request Broker Architecture
CORBA/e	Embedded Real Time CORBA
COTS	Commercial Off The Shelf
CPFSK	Continuous Phase Frequency Shift Keying

Abbreviation	Definition
CVSD	Continuously Variable-Slope Delta modulation
DCD	Device Configuration Descriptor
DLC	Data Link Control
DSP	Digital Signal Processor
DTD	Document Type Definition
FM3TR	Future Multiband Multiwaveform Modular Tactical Radio
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPS	Global Positioning System
ICWG	Interface Control Working Group
ID	Identifier
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronic Engineers
JPA	JTRS Platform Adapter
JTNC	Joint Tactical Networking Center
JTR	Joint Tactical Radio
JTRS	Joint Tactical Radio System
LwAEP	Lightweight Application Environment Profile
MAC	Media Access Control
MILCOM	Military Communications Conference
MIPS	Million Instructions Per Second
MHAL	Modem Hardware Abstraction Layer
MOCB	MHAL On Chip Bus
OE	Operating Environment
OMG	Object Management Group
ORB	Object Request Broker
PIM	Platform Independent Model
POSIX®	Portable Operating System Interface
PSM	Platform Specific Model
RPC	Remote Procedure Control
R-S	Reed Solomon
SAD	Software Assembly Descriptor
SCA	Software Communications Architecture
SCD	Software Component Descriptor
SDR	Software Defined Radio
SPD	Software Profile Descriptor
TCP-IP	Transmission Control Protocol (TCP) and Internet Protocol (IP)
TD	Technical Director
TDMA	Time Division Multiplexed Access
UI	User Interface

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

Abbreviation	Definition
UML	Unified Modeling Language
UOF	Unit of Functionality
WF	Waveform
XML	eXtensible Markup Language