European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification Volume II: Developer's Guide

BAE Ref No: IAWG-ECOA-TR-002
Dassault Ref No: DGT 144475-B

Issue: 2

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: *This specification represents the output of a research programme and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This standard of documentation is recommended as appropriate for limited lab-based evaluation only. Product development based on this standard of documentation is not recommended.*

1 Table of Contents

2 List of Figures

3 List of Tables

No table of figures entries found.

4 Abbreviations

API	Application Programming Interface
ARINC	Aeronautical Radio INC
ASAAC	Allied Standard Avionics Architecture Council
AS	Architecture Specification
ASC	Application Software Component
ASCI	Application Software Component Interface
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
OS	Operating System
POSIX	Portable Operating System Interface
QoS	Quality of Service
SOA	Service Oriented Architecture
SW	Software
UCS	Universal Multiple-Octet Coded Character Set
UDP	Unreliable Datagram Protocol
UML	Unified Modelling Language
XML	Extensible Mark-up Language
XSD	XML Schema Definition

5 Executive Summary

The European Component Oriented Architecture (ECOA) programme represents a concerted effort to reduce development and through-life-costs of the increasingly complex software intensive systems within military platforms by developing a software architecture and paradigm supporting service oriented concepts. Initially, the ECOA Architecture is focussed on supporting mission system software of combat air platforms, both new build (e.g. Unmanned Air Systems) and legacy upgrades. However the ECOA solution is equally applicable to mission system software of land, sea and other air platforms.

This document is a guide to developers of ECOA Applications and ECOA Platforms. It is the second volume of the ECOA architectural specification. This will be up-issued to reflect the finalised ECOA Architecture later in the programme.

As a precursor to this document, the reader is encouraged to consult the Key Concepts Document (Reference 1), Common Terminology (Reference 11) and ECOA Architecture Specification Volume I: Key Concepts (Reference 1) which introduces the ECOA concepts and terms.

This document should be read in conjunction with the ECOA Architecture Specification Volume III: Reference Manuals which provide more detailed descriptions of the Ada, C and C++ Bindings, the ECOA Mechanisms and the Software Interface.

The main parts of this document are:

- Software Development Activities – this section describes the activities entailed in designing and developing ECOA Application Software Components (ASCs) and ECOA Software Platforms.
- Legality Rules – this section identifies the rules to be followed in order that an ECOA ASC or ECOA Platform are consistent with the ECOA Specification.

This document is a Work-In-Progress.

ONLY section 7.3 (Software Platform Development) has been completed and reviewed by the ECOA Programme Partners.

6 Introduction

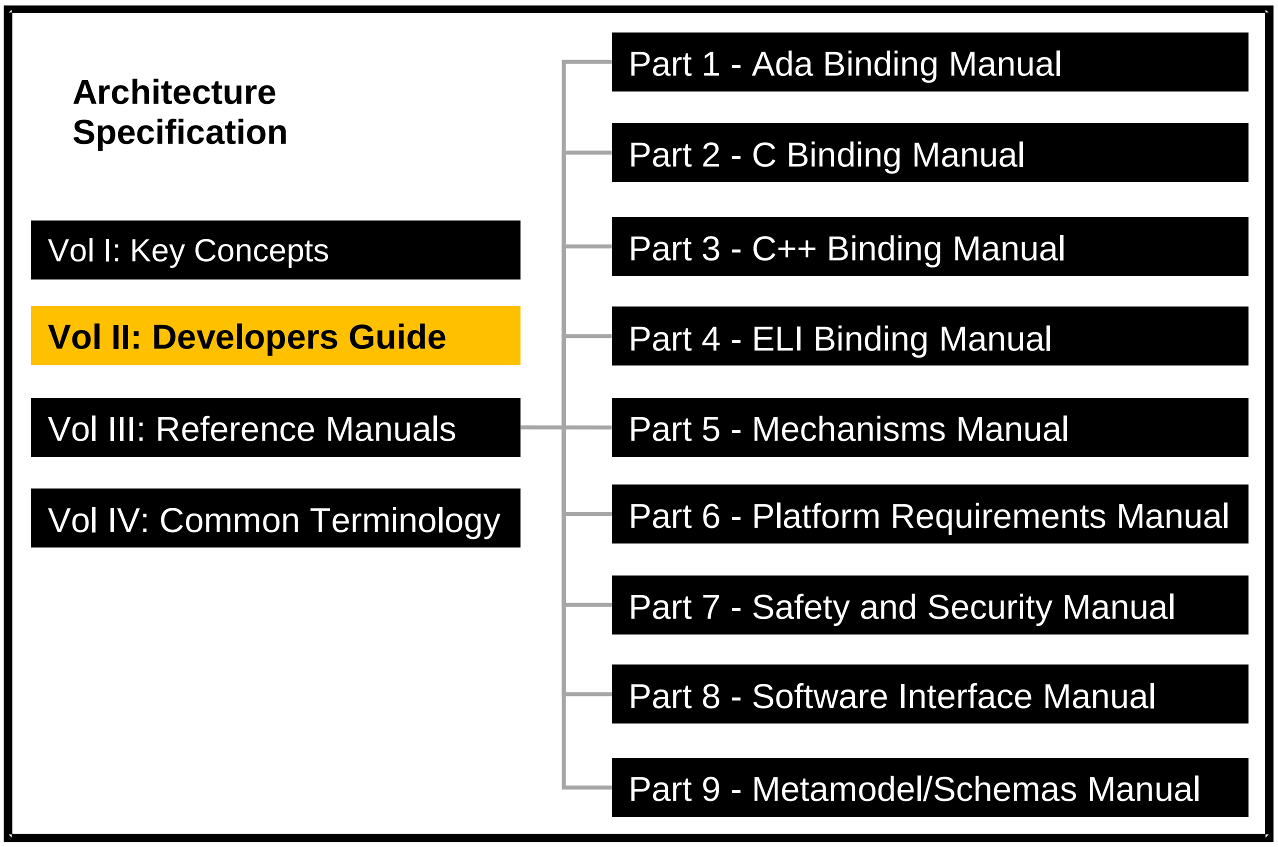

Architecture Specification
Vol I: Key Concepts
Vol II: Developers Guide
Vol III: Reference Manuals
Vol IV: Common Terminology
Part 1 - Ada Binding Manual
Part 2 - C Binding Manual
Part 3 - C++ Binding Manual
Part 4 - ELI Binding Manual
Part 5 - Mechanisms Manual
Part 6 - Platform Requirements Manual
Part 7 - Safety and Security Manual
Part 8 - Software Interface Manual
Part 9 - Metamodel/Schemas Manual

Figure 1 – ECOA Documentation

The Architecture Specification provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA-based system. It is introduced in Key Concepts (Reference 1) and uses terms defined in the Common Terminology (Reference 11). For this reason, the reader should read these documents, prior to this document. The details of the other documents comprising the rest of the Architecture Specification can be found in Section 9

The Architecture Specification consists of four volumes, as shown in Figure 1:

- Volume I: Key Concepts
- Volume II: Developer's Guide
- Volume III: Reference Manual
- Volume IV: Common Terminology

This document comprises Volume II of the ECOA Architecture Specification; it identifies the activities and considerations to be used when developing software using ECOA concepts. The document is intended to be understood by developers who have little knowledge of the ECOA paradigm, so the information has been grouped into activities that should be understood by all.

The document is divided into two main sections:

- a description of the development activities, identifying what considerations the ECOA paradigm requires a developer to do make.

- Identification of rules that must be followed both during development and when producing metadata (xml) which is used when exchanging aspects of your system with third parties.

The intended audience for this document is:

- System Designer
- Software developer implementing an ECOA Application Software Component
- Software developer implementing an ECOA Software Platform
- Software/System Integrator
- Software/System Tester
- Software Team Managers
- Software Process Developer

7 Software Development Activities

This section identifies a series of development activities which are intended to fit into any company process, therefore each section describes 'what' is entailed from each activity but not 'how' these activities are performed. The intention is that the activities are independent of specific Tooling, although the ECOA paradigm lends itself to the use of automation it is perfectly feasible to develop ECOA compliant software by hand.

7.1 Software System Design

Standard software systems comprise of a set of one or more Applications which interact with each other to achieve the functionality of the system. When employing the ECOA paradigm applications are constructed as one or more ECOA Application Software Components (ASCs) and their interactions are described through Services which they provide or require of each other. Software system design under the ECOA paradigm requires the designer to identify the ASCs within the system and the Services that they provide or require. ASCs are then implemented by a developer as Modules, where a Module can be seen as something embodying a thread of program execution.

The recommended way of doing this is to start with the services which the system is required to provide, sub divide those services into areas of responsibility and delegate their provision to specific dedicated ECOA ASCs.

The following sections describe how to use this approach to develop an ECOA software system.

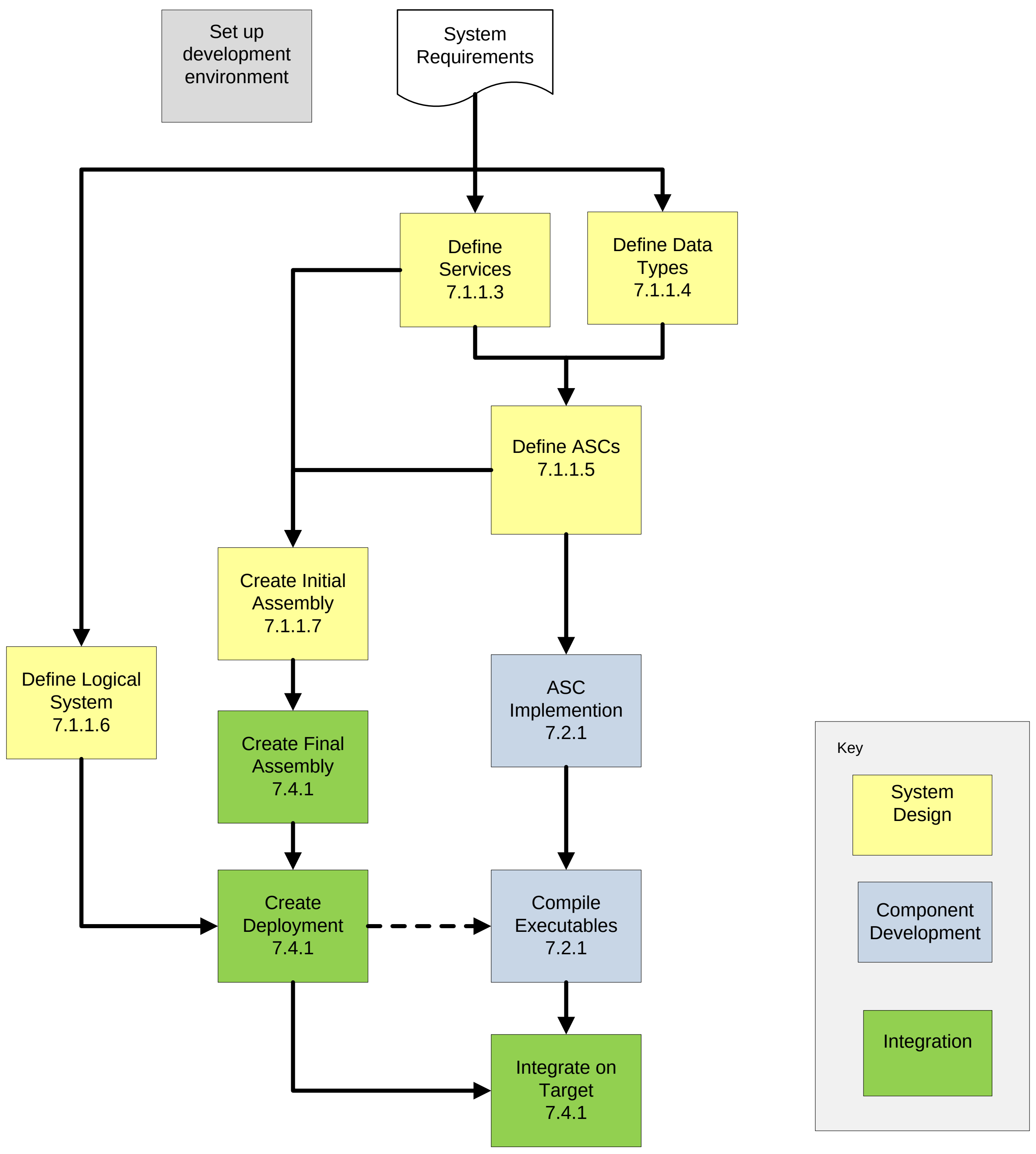

Set up development environment
System Requirements
Define Services 7.1.1.3
Define Data Types 7.1.1.4
Define ASCs 7.1.1.5
Create Initial Assembly 7.1.1.7
Define Logical System 7.1.1.6
ASC Implemention 7.2.1
Create Final Assembly 7.4.1
Create Deployment 7.4.1
Compile Executables 7.2.1
Integrate on Target 7.4.1
Key
System Design
Component Development
Integration

Figure 2 – Software Development Activities

7.1.1 System Designer Activities

Refer to Figure 2.

7.1.1.1 Inputs

- Functional Requirements,
- non-functional complementary requirements.

7.1.1.2 Selection of ECOA Software Platform

The System Designer will need to consider any external constraints on the system, such as interfaces to be supported, performance requirements etc.. These constraints could determine the choice of processing hardware and architecture, which in turn may affect the selection of an ECOA Software Platform provider.

Choice of Scheduling policy – this *might dictate choice of ECOA Platform and/or choice of ASCs?*

In addition Safety and Security issues will need to be considered. Higher certification level ASCs will need to be partitioned, either via hardware or software from lower certification level ASCs. With respect to security the provision of secure communications by the ECOA Platform may be required.

7.1.1.3 Define Services

"Business as usual", analysis of requirements to identify Use Cases for the system.

Successive/iterative decomposition of Use Cases into the services required of the system. This process may be supported by tooling, for instance a customised UML Design tool could be used.

A service is defined as

- One or more operations, which can be any of the following:
  - o VersionedData Operation,
  - o Event Operation
  - o RequestResponse Operation.
- With (currently optional) Quality of Service (QoS) specifications:
  - o Specific attributes dependent on operation type [Ref. Vol III]
  - o Encryption Level
  - o Highest and Lowest Rate – used in early validation and in aiding assessment of thread priority.

7.1.1.4 Define ASCs

It is recommended that appropriate ***reference architecture/specification*** is used to map the identified system services to ASCs.

It is assumed that reference architectures/specifications will be available for the systems to be supported by ECOA ASCs and ECOA Software Platforms. Such specifications are important in order to ensure a level of consistency of service definition and ASC definition/specification. Without such consistency the potential re-use of ASCs will be greatly reduced.

Where there are no available ASCs which provide the required services, or where the provided service operations are not compatible (wrt data types or QoS), consideration will need to be given to the modification of existing ASCs or the development of new ASCs.

If any new ASCs are being developed externally (i.e. by a different company to one performing the system design) or modifications to existing ASCs are being undertaken externally then the appropriate exchange metadata (See Ref. [10]) must be produced for supply to an ASC developer.

For internally sourced ASCs (i.e. those created/maintained by the same company performing the system design) the same data will be required for their development, but if an integrated system/software design toolset is employed then there should be no need to export the Exchange Metadata XML file(s).

A Component is defined by:

- o Properties,
- o ProvidedServices
- o RequiredServices.

A ProvidedService or a RequiredService references a ServiceDefinition shared by ASCs.

In general, an ASC should have at least either a ProvidedService or a RequiredService.

7.1.1.5 Define Logical System

The Logical System is defined, this is a view of the hardware elements. This shows the ECOA Platforms, Computing Nodes and processing cores, together with their links.

7.1.1.6 Create Initial Assembly

The initial assembly describes the structure of an ECOA system, independently of its physical deployment on hardware platforms. It consists of Composites, ASCs and Service Links (also known as Wires).

- A Composite is similar to an ASC in that it has provided or required services, which are ‘promoted’ from the services of its internal ASCs. Composites and the promotion links are a logical concept to master complexity through hierarchical assemblies. They have no existence at technical levels: the assembly schema actually deployed is the one containing only ASCs.

- An ASC is instantiated from a ComponentDefinition within a given system. It has a set of instantiation parameters known as Properties.

- Wires are used to connect ASCs together via provided and required services. Each Wire connects one Provided Service to one Required Service. It is permissible to connect multiple Provided Services to a single Required Service using multiple Wires. In this case the Required service will receive data from all connected Providers.

  The System Designer must indicate a preference for data from a service provider over others when they are connected to the same required service, this is achieved using the Wire’s Rank attribute. The higher the numerical value of rank, the higher the preference for the link. Assignement of a rank value is mandatory.

An additional Boolean attribute **allEventsMulticasted** indicates if all Event operations of the service definitions are sent on this Wire in a systematic way.

Where a modelling tool has been employed for the system design, then early verification of the design may be possible using the Logical System and Initial Assembly. This would verify that the high level system requirements are being satisfied and provide an assessment of the processing power required of the hardware.

7.2 Application Software Component Development

Traditionally software development involves writing application code that conforms to the System Design. The ECOA paradigm is no different in this respect except it specifies a set of rules on how the code of an ASC is implemented. The use of these rules makes it possible to auto-generate the software required to integrate an ASC onto an ECOA Software Platform thereby ensuring that the ASC is portable/reusable on any ECOA Software Platform. This portability is achieved by implementing the ASC code such that it does not directly make use of any underlying functionality of the ECOA Software Platform.

Isolation of ASC code from the underlying hardware and RTOS has been achieved previously using standards such as ASAAC. The ECOA paradigm however also provides for more reuse of the ASC, without any modification, through the standardisation of how ASCs interface with each other.

ASCs may provide Services, and in turn may require Services from other ASCs. ASCs are implemented as ECOA Modules. Services may consist of one or more operations, which are implemented as Module operations. The ECOA specifies the types of operation which are permissible such as: Event, request response or Versioned Data etc.

The ASCs are mapped onto the underlying operating system or middleware through an instance of an ECOA Container. The Container is implemented by software which may be auto-generated by the ECOA tooling.

An ASC’s Modules and the Container code can only communicate via the Application Software Component Interface (ASCI). This is comprised of the following two interfaces:

- **Container Interface**, which is an API available for the Module developer to invoke Container operations which provide access to infrastructure services and interaction with other Modules or ASCs. Module implementations access the Container operations through the API calls defined in the Container Interface.

- **Module Interface** through which the Container invokes Module operations. A Module operation is mapped onto a Module which is defined by a set of Entry Points and their associated software code; Entry Points can be invoked by the Container.

In an ECOA system, all interactions between ECOA Modules rely on three mechanisms: Event, Request-Response and Versioned Data. The behaviour of these is described in Part 5 of Vol III of the Architecture Specification.

7.2.1 ECOA ASC Developer Activities:

The ECOA ASC developer is responsible for taking the ASC specification and developing the ECOA Modules for its implementation. This section describes the steps to be undertaken together with the ECOA specific rules to be followed.

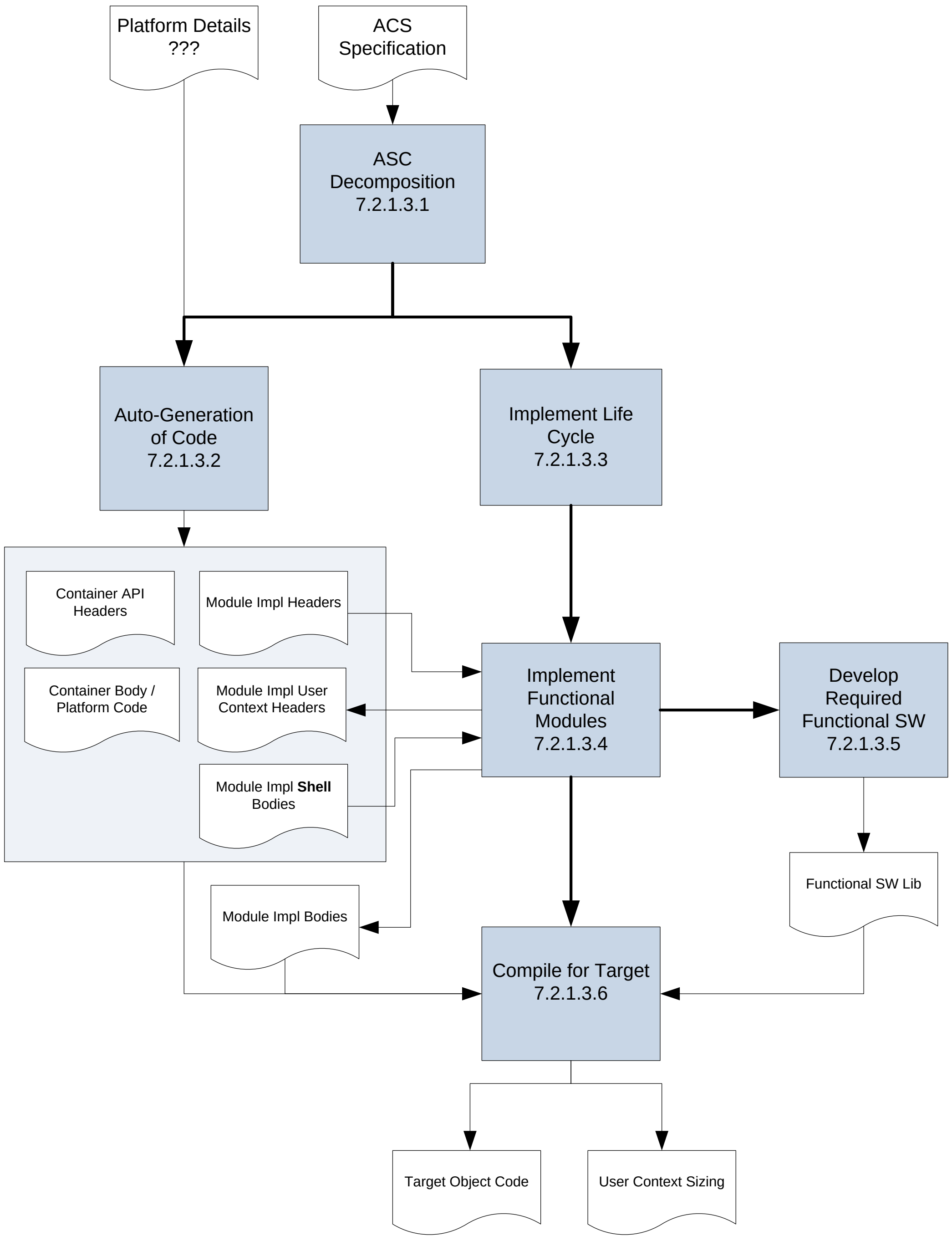

Platform Details ???
ACS Specification
ASC Decomposition 7.2.1.3.1
Auto-Generation of Code 7.2.1.3.2
Implement Life Cycle 7.2.1.3.3
Container API Headers
Module Impl Headers
Container Body / Platform Code
Module Impl User Context Headers
Module Impl Shell Bodies
Implement Functional Modules 7.2.1.3.4
Develop Required Functional SW 7.2.1.3.5
Functional SW Lib
Module Impl Bodies
Compile for Target 7.2.1.3.6
Target Object Code
User Context Sizing

Figure 3– Component Developer Activities

7.2.1.1 Inputs

- Component specification – service behaviour /QoS, together any relevant non-functional complementary requirements.

- Code Generator– supplied by the ECOA Platform provider. This can be used to generate code shells for the ASC implementation, together with the required Container code files. It is expected at least two targets should be supported for the Container file generation, POSIX and the target hardware RTOS/Middleware.

- Test system – a reference system (certified?) on which the developer can perform unit test of the ASC(s).

7.2.1.2 Development Setup

- Mainly “Business as usual” (company preference will likely apply) in the choice of language, e.g. Ada, C or C++. The Container implementation language should not affect the ASC development, since the ECOA Software Platform should contain any shim code that may be required to interface between the ASC implementation Modules and the Container API. Note however not all languages may be supported by the provided Code Generator.

- Configuration of development toolset to support ECOA Naming guidelines, ref AS Vol III. It is expected this would only be needed to be done once and would be reused for all subsequent ECOA programmes.

7.2.1.3 ECOA Specific Considerations

7.2.1.3.1 ASC Decomposition

An ASC is defined by:

- ModuleTypes,
- ModuleInstances,
- TriggerInstances,
- DynamicTriggerInstances
- OperationLinks.

ASC Services are implemented by its Module operations. The selection of operations to be executed by a Module will be primarily determined according to their related functionality. Other considerations include: access to shared data, re-use/sharing of the operation by other ASCs and the schedule-ability of the operation.

An ASC must have at least one Module.

One of the ASC’s Modules must be classified as a Supervision Module (ref.[1] section 8.9.2), and it will be required to support the required ECOA Runtime Lifecycle operations, refer to section 7.2.1.3.3.

7.2.1.3.1.1 Operation Types:

Considerations for service/operation type, [Ref Part 5 of Vol III]:

- **Event** (with and without data)
  This operation type is used for one way/ no wait (for the sender) communication from one Module to another. This operation type can also be used to send Events to more than one Module.

- **Response Request** (where the server response may be immediate or deferred)
  This operation type is used to implement the equivalent of a remote procedure call. The request may carry data ('in' parameters) and the response may also carry data ('out' parameters). The requesting Module behaviour may be specified to either:
  - Synchronous – the requesting Module operation thread waits for the response.
  - Asynchronous – the requesting Module operation thread does wait for a response, instead the response is handled on a callback operation.

- **Version Data**
  This operation type is used to share data between Modules (of the same ASC or different ASCs).
  A requesting Module receives a copy of the data, however it should be noted that the data may not synched with the latest update particularly in a distributed system.
  Optionally a notification can be provided to indicate if the data has been updated.

- **Trigger**
  A Trigger is used to provide an Event at a specified period. These Events can then be used to invoke behaviour at the period of the Trigger at a multiple of its period.

- **Dynamic Trigger**
  A DynamicTrigger is used to provide an Event after a delay after receipt (by the Container) of an input Event. The delay can be specified as a parameter of the Dynamic Trigger.

- Data exchange possible between "compliable items" without container – but on same thread.

7.2.1.3.1.2 Tasking

- Module operations can **only** be invoked by Container threads, Module code is not permitted to use tasking operations, including wait or delay type operations.
- A Module can only be invoked by one Container thread but a Container thread can invoke more than one Module – providing support for multi threaded behaviour.
- Each service operation within an ASC may result in the execution of code in multiple Modules to achieve the desired functional behaviour. The flow of control through the various Modules to support a single service operation's behaviour is known as the **Functional Chain**. Each service operation has QoS attributes such as response time and minimal inter-arrival time which define their temporal performance requirements or guarantees.
- Dependency on Container code – ECOA Module is passive. (Check definition of ECOA Module – definition "unit of deployment" – one or more compliable items)
- Use of explicit tasking operations is forbidden, where the following is required:
  - Periodic operations – enabled through the use of Trigger operations. These have a fixed duration assigned during integration.

    - Timeout behaviour - enabled through the use of Dynamic Trigger operations. These may have a variable duration assigned at execution time.
    - Multi-tasking – enabled by a Container thread invoking more one Modules.
- Deriving the priority for a Container thread *using Module Operation Deadline– is this a developer issue or an integrator issue?*

7.2.1.3.1.3 Data Storage

- Considerations for data storage, Version data. *Sharing state data between multi-threads – i.e. >1 Module accessing same data – is this permitted?*
- Strategy for handling large data sets (i.e. Operation on a database ASC)
  Is TBA

7.2.1.3.2 Auto-Generation of Code

Following definition of the Modules, their Operations and inter Module links, it is advisable that the ECOA Platform provider's Code Generator is used to generate the following:

- Container API headers
- Container Body/Platform Code
- Module Implementation Headers
- Module Implementation User Context Headers
- Module Implementation shell Bodies

The ASC software developer can then begin to populate the Module implementation shell bodies.

It is expected that the ECOA software provider's Code Generator should support both the reference Linux ECOA Platform in addition the target ECOA Software Platform. Initial code generation is likely to be for the Linux platform to support incremental integration and testing.

7.2.1.3.3 Implement Life cycle:

7.2.1.3.3.1 ECOA ASC and Module life cycles

Module Lifecycle
Managed by the Container, ASC & Module states
There must be a Supervision Module (Reference 1 section 8.9.2).
Service availability – include guidelines for declaring availability – *expect this to be dependent in part on the ASC's functional lifecycle as well* TBA

7.2.1.3.3.2 ASC Functional life cycle

Functional lifecycle – there is a dependency with system lifecycle
Managed by the Supervision Module, possibly under direction of a hierarchy of Supervising/Managing ASCs.
Logging requirement in order to support System Integration – split between what the Container can do and what the ASC dev needs to do...
Note: API for LogInfo is only for message text.
Initialisation/Life cycle – impact on version data and other operations.

7.2.1.3.4 Implement Functional Modules

Decomposition, Module structure use of functional code libraries.
Instances of - (Module can be instanced more than once, where each instance can retain its own state)
Initialisation of Module using property data?
Module naming/identification
Assurance level considerations

7.2.1.3.4.1 Data Typing

- How to create complex data type from ECOA basic types.
  Operation types – either basic or specific type definitions.

7.2.1.3.4.2 Health monitoring

- Mainly business as usual – need to expand on features unique to ECOA – TBA
- Fault reporting guidelines, incl sequence diagrams
  Consider describing the scope of remedial actions expected by application SW.

7.2.1.3.5 Develop Required Functional Software

- Mainly business as usual
- This is software which is called directly from an ECOA Module
- Algorithmic code such as navigation, maths or graphics

7.2.1.3.6 Compile for Target

Dependent on ECOA Software Platform selected for the Auto-generate code....

7.2.1.3.7 Insertion Policy

- Assessing stack/heap allocations required -> insertion policy
  Enforcing a "static" memory model. – rule???
- Module properties – why and how. *(they be inherited from ASC properties – check ASv5 errata)*
- Calculation of Module Deadline
- Assessment of Service/Operation QoS being provided
- Assessment of Module priority from a Component Developer POV
- Processing platform – performance requirements relative to reference platform
- RAM/EPROM requirements
- Software Library support required
- RTOS requirements
- Assurance level

7.2.1.4 Component Validation

A **ModuleBehaviour** allows describing characteristics of a **ModuleImplementation** item, which will be used for deployment and early verification analyses (for example: consistency with the ASC level behaviour). It mainly gives a decomposition of Module treatments, allowing to assess CPU resources needs.

A **ModuleBehaviour** is composed of a **ModuleConfiguration** and a set of **entryPoints**.

7.2.1.5 Module(s) tested on reference hardware.

TBA

7.2.1.6 Outputs

- Target object code Library, Insertion policy/User Context Info, Behavioural description, Safety&Security details-TBA.

7.3 Software Platform Development

The term Software Platform is used to describe both the middleware upon which the ECOA applications are running (e.g. ASAAC, ARINC 653, POSIX) and also the Platform Integration Code, which provides the ECOA Application Software Components with the necessary resources and mechanisms to function (e.g. platform management, node management, Protection Domain management).

The following sections describe the functions to be implemented by an ECOA Software Platform, along with some guidance on how certain aspects may be implemented. ECOA does not mandate how platforms or mechanisms are implemented; rather it describes the behaviours required and only specifies the interfaces. The intent is to avoid being too prescriptive about the ECOA Platform design, and thereby preserve the freedom of platform developers to implement their ECOA Software Platform in the most optimal way for their specific hardware, and low level software.

Figure 4 (following) illustrates a purely functional view of the elements which constitute an ECOA Software Platform.

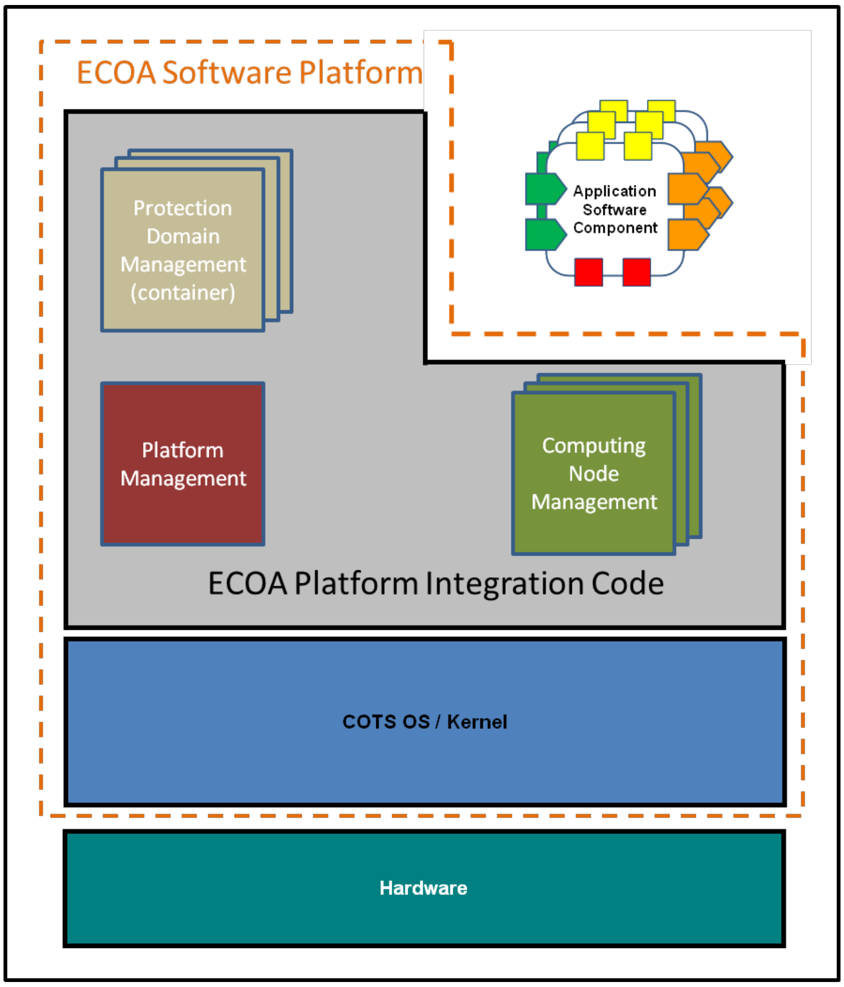

ECOA Software Platform
Application Software Component
Protection Domain Management (container)
Platform Management
Computing Node Management
ECOA Platform Integration Code
COTS OS / Kernel
Hardware

Figure 4 - Functional View of an ECOA Software Platform

The functional elements shown in Figure 4 can be mapped onto a deployment viewpoint.

Figure 5 shows the Protection Domain Management functional element from a deployment viewpoint. Here, a single Protection Domain can provide functionality to manage the Protection Domain, alongside Container functionality to manage/support one or more ECOA Application Software Components. A Protection Domain provides spatial and potentially temporal partitioning.

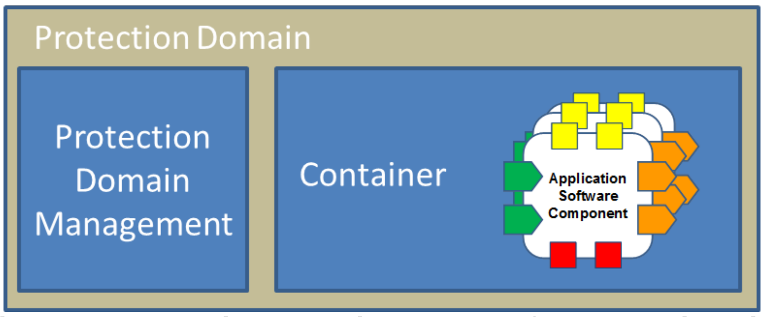


Figure 5 - Protection Domain Level Platform Functionality

In the same manner, Figure 6 shows the Computing Node Management functional element from a deployment viewpoint. Here, a single Computing Node can perform functionality to manage the node, along with one or more Protection Domains. The Computing Node Management functionality may be distributed between one or more Protection Domains, or be contained in a Protection Domain itself.

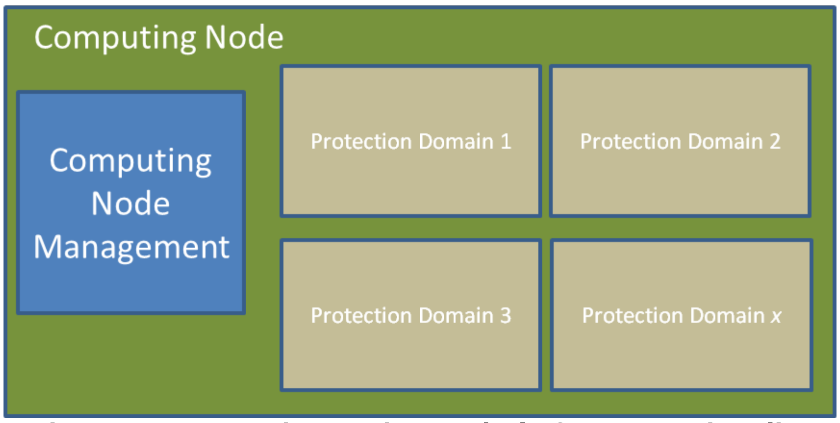


Figure 6 - Computing Node Level Platform Functionality

Finally, Figure 7 shows the Platform Management functional element from a deployment viewpoint. Here, a single Platform can perform functionality to manage the platform, alongside functionality to manage/support one or more Computing Nodes. The Platform Management functionality will necessarily be hosted on a Computing Node; however this may be distributed between the functional Computing Nodes, or be contained in a single Computing Node with other functions, or by itself.

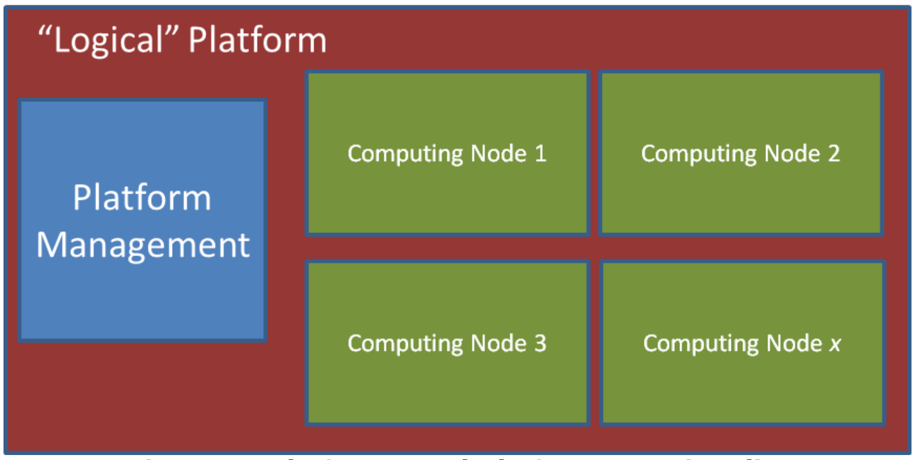


Figure 7 - Platform Level Platform Functionality

7.3.1 ECOA Software Platform Requirements

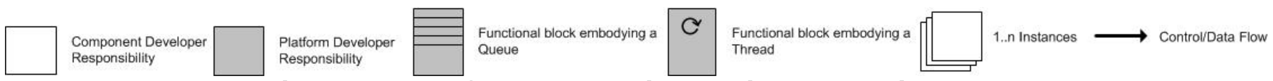
The requirements for an ECOA Software Platform are derived from the mechanism behaviour specified in the Mechanisms Reference Manual (Reference 6), and can be found in the Platform Requirements Reference Manual (Reference 7)

7.3.2 ECOA Software Platform Integration Code

Although the ECOA Platform Integration Code will be unique for an underlying COTS OS/Kernel, programming language and platform provider; it is possible to have a generic design.

Figure 9 shows a possible functional breakdown of the generic elements which are required, and Figure 10 shows the detail within the Component Implementation functional element. Figure 8 gives a key to be used for these diagrams.

Note: Whilst the ECOA Specification allows for the distribution of the Modules of ASCs across Protection Domains this aspect has not yet been developed by the ECOA Programme, and is not accommodated by this possible functional breakdown.


Component Developer Responsibility
Platform Developer Responsibility
Functional block embodying a Queue
Functional block embodying a Thread
1..n Instances
Control/Data Flow
Figure 8 – Platform Integration Code Key to Diagrams

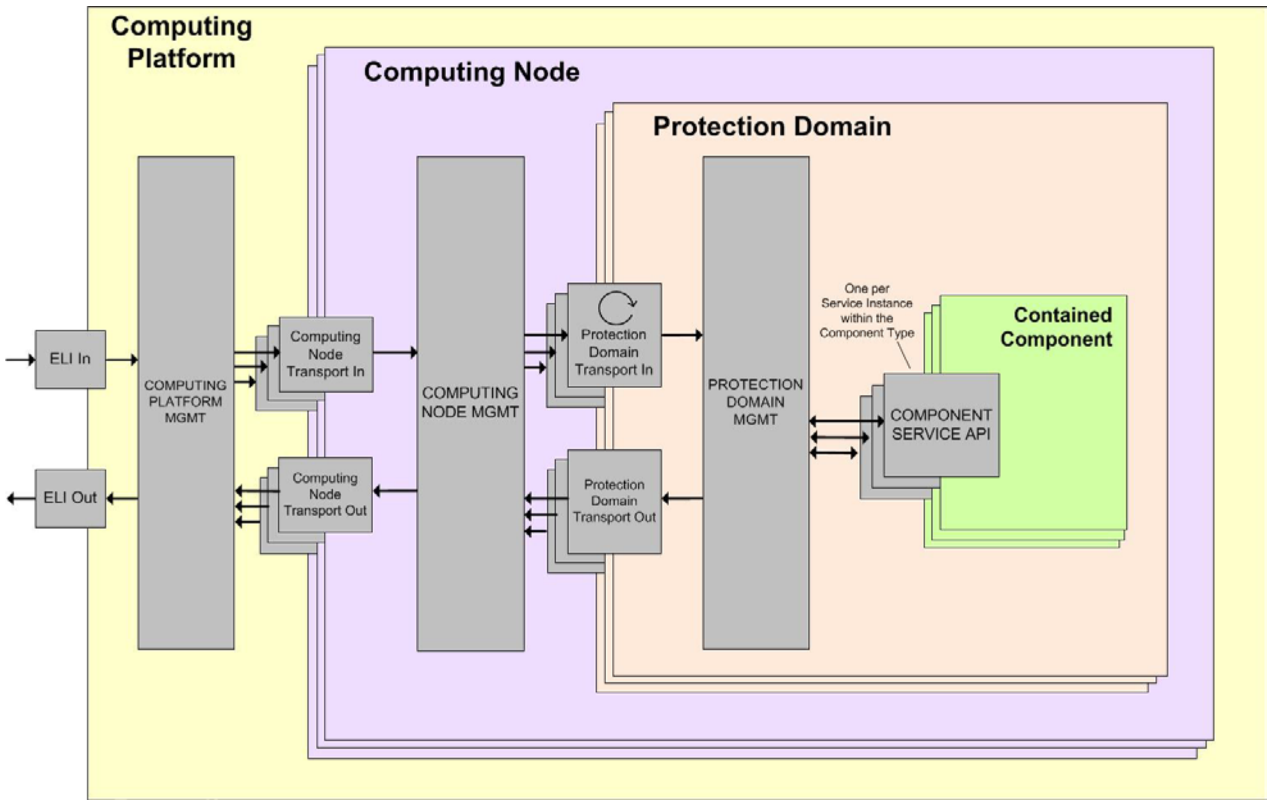

Computing Platform
Computing Node
Protection Domain
ELI In
ELI Out
COMPUTING PLATFORM MGMT
Computing Node Transport In
Computing Node Transport Out
COMPUTING NODE MGMT
Protection Domain Transport In
Protection Domain Transport Out
PROTECTION DOMAIN MGMT
One per Service Instance within the Component Type
Contained Component
COMPONENT SERVICE API

Figure 9 – Platform Integration Code Functional Design

7.3.2.1 Computing Platform

Within the Computing Platform boundary in Figure 9, the following functional elements are required:

7.3.2.1.1 Computing Platform Mgmt

The Computing Platform Mgmt is responsible for the management of the Computing Nodes within the same ECOA Platform, including power-up & initialization, fault management, configuration management and communications management. It routes operations between Computing Nodes within the same ECOA Platform and, if required, outside the ECOA Platform (using the ELI In/Out functionality). The method of routing is defined to maintain inter-operability; the ECOA Logical Interface (ELI) (see Reference 5) must be used.

In terms of internal-platform communication, the ELI message may need to be transformed into the format required by the ECOA Software Platform.

7.3.2.1.2 ELI In

The ELI In is responsible for receiving ELI messages directed to the ECOA Platform. The ELI message format is defined by the ECOA Specification (see Reference 5). The responsibilities of the ELI In can vary depending upon the transport medium used and the internal-platform communication methods. For example, if UDP is used, large ELI messages (greater than the maximum size of a UDP payload) are fragmented, so the ELI In would be responsible for reassembling a number of UDP packets into a complete ELI message.

7.3.2.1.3 ELI Out
The ELI Out is responsible for sending ELI messages to other ECOA Platforms. The ELI message format is defined by the ECOA Specification (see Reference 5). The responsibilities of the ELI Out can vary depending upon the transport medium used and the internal-platform communication methods. For example, if UDP is used, large ELI messages (greater than the maximum size of a UDP payload) will require fragmenting, and the ELI Out would be responsible for the fragmentation of the message.

7.3.2.2 Computing Node
Within the Computing Node boundary in Figure 9, in addition to the Protection Domains the following functional elements are required:

7.3.2.2.1 Computing Node Mgmt
The Computing Node Mgmt is responsible for the management of the Protection Domains within the same Computing Node, including loading, initialization, fault management, configuration management and communications management. It routes operations between Protection Domains within the same Computing Node and, if required, outside the Computing Node (using the Computing Node Transport In/Out functionality).The method of transport is not defined and is implementation specific.

7.3.2.2.2 Computing Node Transport In
The Computing Node Transport In is responsible for managing routing of incoming operations to the Computing Node Mgmt from other parts of the system (including other Computing Nodes). The method of transport is not defined and is implementation specific.

7.3.2.2.3 Computing Node Transport Out
The Computing Node Transport Out is responsible for managing routing of outgoing operations from the Computing Node Mgmt to other parts of the system (including other Computing Nodes). The method of transport is not defined and is implementation specific.

7.3.2.3 Protection Domain
Within the Protection Domain boundary in Figure 9, in addition to the Contained Component the following functional elements are required:

7.3.2.3.1 Protection Domain Mgmt
The Protection Domain Mgmt is responsible for the management of the Contained Components within the same Protection Domain, including loading (if necessary), initialization, fault management, configuration management and communications management. It routes operations between contained ASCs within the same Protection Domain and, if required, to other parts of the system (using the Protection Domain Transport In/Out functionality). The method of transport is not defined and is implementation specific.

It is possible to use the ELI message format internally to a Protection Domain; however, due to the nature of the ELI message definition (big-endian and tightly packed data), this may not be desirable.

7.3.2.3.2 Protection Domain Transport In
The Protection Domain Transport In is responsible for managing routing of incoming operations to the Protection Domain Mgmt from other parts of the system (including other Protection Domains). The method of transport is not defined and is implementation specific.

7.3.2.3.3 Protection Domain Transport Out

The Protection Domain Transport Out is responsible for managing routing of outgoing operations from the Protection Domain Mgmt to other parts of the system (including other Protection Domains). The method of transport is not defined and is implementation specific.

7.3.2.4 Contained Component

Within the Contained Component boundary in Figure 9, the following functional elements are required, and shown in Figure 10. Note that for any Contained Component there may be many Module Implementations and there may be multiple instantiations (Module instances) of any given Module Implementation.

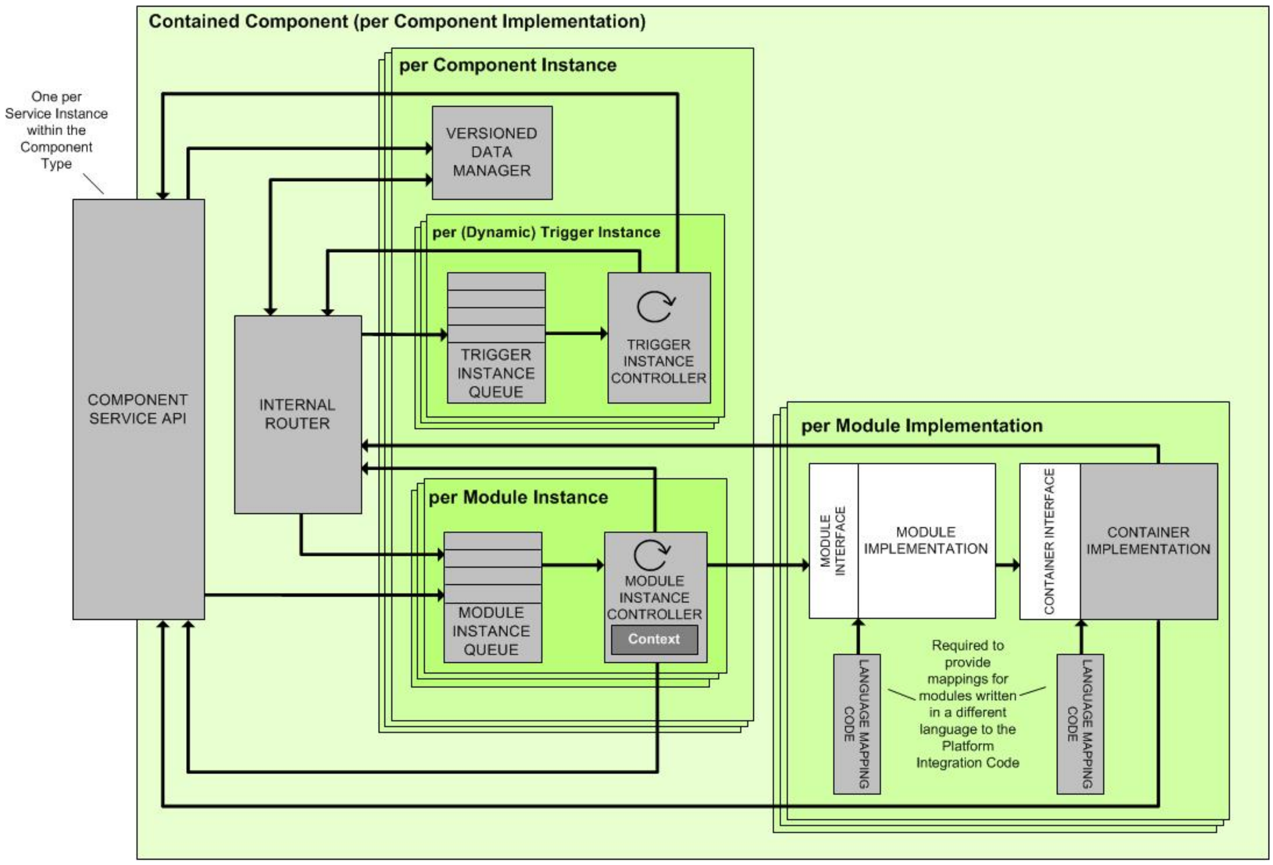

Contained Component (per Component Implementation)
per Component Instance
One per Service Instance within the Component Type
VERSIONED DATA MANAGER
per (Dynamic) Trigger Instance
TRIGGER INSTANCE QUEUE
TRIGGER INSTANCE CONTROLLER
COMPONENT SERVICE API
INTERNAL ROUTER
per Module Implementation
per Module Instance
MODULE INSTANCE QUEUE
MODULE INSTANCE CONTROLLER
Context
MODULE INTERFACE
MODULE IMPLEMENTATION
CONTAINER INTERFACE
CONTAINER IMPLEMENTATION
LANGUAGE MAPPING CODE
Required to provide mappings for modules written in a different language to the Platform Integration Code
LANGUAGE MAPPING CODE

Figure 10 – Platform Integration Code Component Implementation

7.3.2.4.1 Module Implementation

The Module Implementation shown in Figure 9 represents the functional Module code provided by an ASC developer (typically delivered as a pre-compiled binary object). The functionality provided by the Module will be invoked by its associated Module Instance Controller using the ECOA defined Module Interface.

The Module Implementation code can also invoke Module Implementation specific Container operations, defined in section 7.3.2.4.2 using the ECOA defined Module Implementation specific Container Interface.

7.3.2.4.2 Container Implementation

The Container Implementation comprises the code implementing the ECOA defined Container Interface for the specific Module Implementation, against which the ASC developer has compiled.

The implementation of this interface must first identify the calling Module instance (typically by inspecting the Container private context) and then either invoke the Internal Router or ASC Service API (or both) depending upon the connectivity defined by the ASC implementation specification.

As an example, if a Module instance performs an Event send operation that is connected to another local Module instance, the Internal Router would be invoked to route the Event to the correct Module Instance Queue (determined from the senders' Module instance and senders' ASC instance). Equally, if a Module instance initiates an Event send operation that is connected to a service, the ASC Service API would be invoked to route the Event to the receiving ASC instance / required service instance (determined from the senders' ASC instance and the wiring within the Assembly).

Although the Container Implementation could be used to directly queue operations to local Module instances, the use of an Internal Router removes the need for ASC instance specific logic within the Container Implementation; therefore allowing the Container Implementation to be generic between multiple instantiations of an ASC.

7.3.2.4.3 Module Instance Controller

The Module Instance Controller's primary responsibility is to process its Module Instance Queue and invoke any queued operations on the Module instance it is responsible for controlling. Each Module instance defined within the ASC should have its own Module Instance Controller (and consequently a corresponding Module Instance Queue).

The Module Instance Controller will provide the (implementation specific) logic for determining when to invoke an operation on a Module instance (based upon the scheduling policy e.g. rhythmic/reactive).

The Module Instance Controller will also provide the logic for determining when to invoke an operation or reject an operation (based on the Module Runtime Lifecycle state of the Module).

In addition, the Module Instance Controller will hold the current Module Runtime Lifecycle state of the Module instance (IDLE/READY/RUNNING).

7.3.2.4.4 Module Instance Queue

Each Module instance will require a Module Instance Queue in order to queue pending operations written to by the Componnet Service API and Internal Router. The Module Instance Queue will be read from, by its corresponding Module Instance Controller.

7.3.2.4.5 Trigger Instance Controller

The Trigger Instance Controller's primary responsibility is to send Event operations, at the time intervals defined for the Trigger Instance/Dynamic Trigger it implements. Each Trigger Instance or Dynamic Trigger defined within the ASC should have its own Trigger Instance Controller (and consequently a corresponding Trigger Instance Queue).

The Trigger Instance Controller will process its corresponding queue to invoke Module state changes or accept Trigger Instance lifecycle commands (in the case of Dynamic Triggers).

7.3.2.4.6 Trigger Instance Queue

Each Trigger Instance will require a Trigger Instance Queue in order to queue pending commands. The Trigger Instance Queue will be written to by the Internal Router, and read from, by its corresponding Trigger Instance Controller.

7.3.2.4.7 Internal Router

The Internal Router is responsible for routing all operations that have a source and destination within the boundary of the Contained Component. The Internal Router will be unique to each ASC implementation, as the connectivity and routing is dependent upon the ASCs implementation specification. However, as the ASC can be instantiated multiple times in different deployments, the Internal Router is specific to a deployment (in that it will need to determine the callers' ASC instance in order to route to the appropriate Module instance).

The Internal Router will provide the ability to route operations between Module instances (via the relevant Module Instance Queue or Versioned Data Manager) including:

- Event sends (where the receiver is a local Module instance)
- Request sends (sync/async) (where the server is a local Module instance)
    - NOTE: for a synchronous call the module instance thread would be blocked by the Internal Router awaiting the response
- Response sends (where the client is a local Module instance)
- Versioned Data writes (always required in order to update the ASC instance "local" copy of the Versioned Data)
- Versioned Data reads (always required in order to get the latest ASC instance "local" copy of the Versioned Data)
- Versioned Data update notifications (where a reader is a local Module instance and requires notification of update)
- Get Properties (always required in order to get the Properties for the ASC instance/Module instance)

7.3.2.4.8 Versioned Data Manager

The Versioned Data Manager is responsible for storing the latest version of the ASC instance "local" copy of the Versioned Data, and allocating versions of the data when requested by a Module (through the Internal Router). An update to versioned data is notified to the Internal Router to allow an update notification to be generated if required.

A Versioned Data Manager should exist for every Versioned Data in an ASC instance.

A more detailed description of Versioned Data Management is given in section 7.3.8.

7.3.2.4.9 Component Service API

The Component Service API is responsible for managing communication across the Contained Component boundary. There should be a Component Service API for every Service Instance of each ASC type, and would contain functionality for all the service operations. Each Component Service API is capable of supporting multiple instances of that Component Type. The wiring of services in the Assembly provides the routing information for the service operations.

7.3.2.4.9.1 Provided Services

For provided services, the Component Service API should contain functionality to route operations from the Contained Component including:

- Event sends (where the Event is "sent by provider")
- Response sends
- Versioned Data writes

For provided services, the service API should contain functionality to route operations to the Contained Component including:

- Event received (where the Event is "received by provider")

- Request received

7.3.2.4.9.2 Required Services

For required services, the Component Service API should contain functionality to route operations from the Contained Component including:

- Event sends (where the Event is “received by provider”)
- Request sends (async/sync)
    - o NOTE: for a synchronous call the module instance thread would be blocked by the Component Service API awaiting the response

For required services, the Component Service API should contain functionality to route operations to the Contained Component including:

- Event received (where the Event is “sent by provider”)
- Response received
- Versioned Data updates

7.3.3 Service Availability

Service availability is the mechanism by which ECOA provides the dynamic behaviour often found within service oriented architectures. The mechanism allows ASCs to determine if their required services are available, and determine and inform the rest of the system whether their provided services are available. The Platform Integration Code is required to maintain and distribute information regarding the availability of the services such that ASCs may access it as required. In order to facilitate this, the ELI specification (see Reference 5) defines messages used by the ECOA Platform to inform other ECOA Platforms of changes in service availability. By using these messages the ECOA Platform can update its local view of the state of services across the system, and contribute its own service states to other ECOA Platforms.

7.3.4 Internal Platform Communications

In order for all nodes and Protection Domains within an ECOA Platform to correctly route any service requests, and to have visibility of the availability of services they require, an ECOA Platform provider will have to implement an internal platform distribution mechanism. This internal distribution mechanism may also use the ELI message definitions; however this is not mandated, and may not represent the optimal way of implementing this for any specific ECOA Platform.

It is left to the ECOA Platform implementer to determine the optimal method of internal communications based upon the details of their ECOA Platform. The ECOA Platform is required to communicate with other ECOA Platforms using the ELI definition (Reference 5) to ensure interoperability.

7.3.5 Module Context

All Modules have an associated Module Context, which comprises information used by the ECOA Software Platform to manage the instances within the system, along with a user defined part that allows per instance state to be maintained.

The implementation of the Module Context varies with the language used to implement the Module, but for certain languages a reference to the Module Context is passed as a parameter to Module operation calls. In these cases the Module Context is defined as a structure containing the user defined part, and a ‘platform hook’ that may be used by the ECOA Software Platform to manage the Module.

The ‘platform hook’ is defined in the interface (always), however it is not mandated that the ECOA Software Platform make use of the ‘platform hook’ to manage the Module. An ECOA Software Platform may use the ‘platform hook’ a way of accessing the ECOA Software Platform related information directly, since the Module Context is passed to the ECOA Software Platform through any Container operation calls. However since the ‘platform hook’ is visible to the Module implementation, it may be accessed/modified, causing unpredictable behaviour. If required, a ECOA Software Platform implementation may use other means to identify the Module instance invoking Container operations (e.g. by using task/thread identifiers), and leave the ‘platform hook’ unused (probably defaulted to zero).

7.3.6 Scheduling

The low-level thread scheduling policy is not defined or mandated by ECOA, and is left to the System Integrator to define. Within an ASC implementation, the XML defines the time by which any operation on the Module will have completed, which may be used by the System Integrator to determine the scheduling parameters (e.g. priority) to be used in the deployed system. Within the deployment XML this priority information can be captured for every Module instance and Trigger Instance, which can then be used by the scheduling system.

Module operation scheduling policy however is defined by the ECOA Specification, which defines two policies: the Reactive Execution Model; and the Rhythmic Execution Model. See Reference 11.

7.3.7 Partitioning

Although ECOA does not mandate any particular partitioning scheme for ASCs, it provides the concept of Protection Domains for use where isolation in space and (maybe) time is required (see Reference 1 section 8.10). The normal model for a system built using ECOA ASCs is that an ASC would reside in its own Protection Domain. This model could support mixing ASCs with different levels of assurance within the same system, and ultimately on the same Computing Node. Whether this is feasible depends upon the level of assurance being sought, and the nature of the safety requirement.

Whilst this is the normal model, it is also possible to deploy multiple ASCs within the same Protection Domain. This would usually be done for reasons of performance, or simplicity. Components within the same Protection Domain necessarily will have more closely coupled communications paths, and hence provide a faster response, where this is required.
In addition, for rapid prototyping and deployment in non-critical environments it may be appropriate to deploy multiple ASCs within the same Protection Domain, as it will reduce the overheads of building a more complex distribution mechanism.

7.3.8 Versioned Data Management

Whenever a Module performs a read or write request on a Versioned Data item, the ECOA Software Platform is required to allocate an instance of the data item that will not be independently updated whilst it is being used by the calling Module. In order to achieve this, an ECOA Software Platform may dynamically allocate memory (e.g. from heap), and use this to hold the version of the data. Once the Module has finished using the version of the data, the ECOA Software Platform can release the memory back to the pool.

Whilst dynamic allocation/de-allocation fulfils the requirements of the ECOA Software Platform and provides a useable capability to the Modules within the system, it may not be appropriate for all scenarios, as the temporal and spatial properties may be difficult to capture. In cases where it is desirable to have a statically allocated memory map, the ECOA Software Platform would be required to define a number of pre-allocated memory areas for use by Versioned Data

Management. It is also possible to implement these mechanisms using fixed time algorithms to ensure there is no undesired variability in timing.

Another area of Versioned Data Management that the ECOA Software Platform is required to perform is the distribution of updates to all ASCs/Modules that require the data (subscribers). Given the distributed nature of ASCs, and any Versioned Data service operations, it is the responsibility of the ECOA Software Platform to ensure that any updates (publish) of a Versioned Data is distributed to all Protection Domains/Computing Nodes/ECOA Software Platforms that require it. Whether each Protection Domain has a repository, or whether there is a central repository for each Computing Node or ECOA Software Platform is an implementation decision, and may be influenced by the underlying capabilities of the ECOA Software Platform.

7.3.9 Module Lifecycle Management

The ECOA Mechanisms Reference Manual (Reference 6) describes the Runtime Lifecycles of both the ASCs and Modules they contain. Component Runtime Lifecycles are managed by the Supervision Module (ref.[1] section 8.9.2) within an ASC, whereas the Module Runtime Lifecycles are managed by the ECOA Software Platform.

It is the responsibility of the ECOA Software Platform to initialise all Modules prior to moving the ECOA Supervision Modules within each ASC to a running state. The ECOA Supervision Modules then use Container operations provided by the ECOA Software Platform to initialize and start/stop other Modules within the ASC.

The ECOA Software Platform manages the state of each Module instance, and based upon that state, determines whether messages are queued into the appropriate Module Instance Queue. The ECOA Software Platform may also change the state of a Module based upon the detection of errors, and notify its Supervision Module of that change.

7.3.10 Component Lifecycle Management

Component Runtime Lifecycles are managed by the Supervision Module (ref.[1] section 8.9.2) within an ASC according to the (optional) ECOA Specification Component Lifecycle Service definition (Reference 1 section 8.9.1, Reference 6 section 8.1). Where this is the case, the Component Definition shall specify the Component Initial State.

7.3.11 Platform Platform Level ELI

In order for multiple ECOA Platforms to coordinate their initialisation and start-up, a set of ECOA Platform level ELI messages have been defined (see Reference 5). These messages include notifications of changes to ECOA Platform status, service availability and configuration. By using these messages and following the behaviours defined in Reference 5, a ECOA Platform can discover other ECOA Platforms, inquire of available services, request the status of published Versioned Data, and be notified of any changes to these.

8 Legality Rules

The following section identifies the legality rules that a developer 'must' do to ensure that the Application or Software Platform is consistent with the ECOA standard. It does not preclude the use of new or existing company-level development standards for the design of the Application or Software Platform.

8.1 Naming Rules and Conventions

When used within a Module, each of the above abstract APIs is instantiated at Module level per operation. Each operation declared in a Module will necessarily cause the definition of one or more functions used in that Module. As a consequence:

- Each ASC implementation name must be unique within its host Protection Domain
- Each ASC instance name must be unique within the assembly schema
- Each Module implementation name must be unique across the assembly
- Each Module instance name must be unique within the ASC implementation
- Each Module implementation name must be unique within its host Protection Domain
- Each operation name must be unique within each Module definition
- Operation and Module names must follow the naming conventions for identifiers used in the most common programming languages: a name being a sequence of letters, figures and underscores, beginning with a letter.
- The order of Operation Parameters in the Component Definition and Implementation must match the order declared in the Service Definition.

8.2 Module Instances and Context

It is required that the same implementation of a Module can be instantiated several times, possibly within the same Protection Domain, without causing any symbol collision. To achieve this requirement, it is expected, for example, that the implementer of a C or C++ Module would not use any static (either global or local) variables within the Module (except for constants). To this end, Modules are coded with instance specific data blocks referred to as the "Module context".

The purpose of this "Module Context" is to hold all the private data that will be used:

- by the Container and the ECOA infrastructure to handle the Module instance (infrastructure-level technical data),
- by the Module itself to support its functions (user-defined local private data).

The use and the declaration of the "Module Context" structure may be adapted for each language binding.

For non-OO languages, the “Module Context” will be represented as a structure that will hold both the user local data (called “User Module Context” in the binding sections) and all the infrastructure-level technical and specific part of “Module Context” (such technical data won’t be specified in this document as they are implementation dependant). For this reason, the Module Context may be generated by the ECOA infrastructure within the Container Interface Header, and be extended by a user defined "User Module Context" structure.

With OO languages, the Module will be instantiated as an object of a Module Implementation class declared by the user; its associated Container will be associated to an instance of an ECOA-generated Module Container class. All the "User Module Context" will be declared within the user Module Implementation class as its private attributes and accessed through public helper methods; all the infrastructure-level technical data will be declared by the ECOA-infrastructure within the corresponding (generated) Module Container class. In addition, the entry-points declared in the Container Interface are represented as methods of the Container object, so the Module instance object must have access to its corresponding Container object to enable it to call these methods. This can be done by passing a pointer of the Container object as a parameter of the constructor of the Module Implementation class. The Module instance object will use a private attribute to store this pointer to the Container object for future use.

9 References

Ref.	Document Number	Version	Title
1.	IAWG-ECOA-TR-001	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume I Key Concepts
2.	IAWG-ECOA-TR-003	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 1: Ada Binding Reference Manual
3.	IAWG-ECOA-TR-004	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 2: C Binding Reference Manual
4.	IAWG-ECOA-TR-005	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 3: C++ Binding Reference Manual
5.	IAWG-ECOA-TR-006	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 4: ELI and Transport Binding Reference Manual
6.	IAWG-ECOA-TR-007	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 5: Mechanisms Reference Manual
7.	IAWG-ECOA-TR-008	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 6: Platform Requirements Reference Manual
8.	IAWG-ECOA-TR-009	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 7: Approach to Safety and Security Reference Manual
9.	IAWG-ECOA-TR-010	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 8: Software Interface Reference Manual
10.	IAWG-ECOA-TR-011	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 9: Metamodel and XSD Schemas Reference Manual
11.	IAWG-ECOA-TR-012	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume IV Common Terminology