European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification Part 1: Concepts

BAE Ref No: IAWG-ECOA-TR-001
Dassault Ref No: DGT 144474-F

Issue: 6

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: *This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development should rely on the DefStan or BNAE publications of the ECOA standard.*

Contents

Figures

Tables

0 Introduction

0.1 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.

ECOA® aims to facilitate rapid system development and upgrade to support a network of flexible platforms that can cooperate and interact, enabling maximum operational effectiveness with minimum resource cost. ECOA® provides a software architectural approach to facilitate this.

ECOA® is primarily focused on facilitating the construction of mission system software for combat air platforms; however the ECOA® solution is equally applicable to mission system software of land, sea and non-combat air platforms. The incorporation of legacy software into ECOA is also supported.

ECOA® is documented in its eleven part Architecture Specification.

0.2 Main Introduction

The Architecture Specification provides the specification for creating ECOA®-based systems. It describes the standardised programming interfaces and data-model that allow developers to produce ECOA® components and construct ECOA®-based systems. It uses terms defined in the Definitions (Architecture Specification Part 2). The details of the other documents comprising the rest of this Architecture Specification can be found in Section 3.

This document is Part 1 of the Architecture Specification; it acts as an introduction to ECOA®.

The document is structured as follows:

- Section 6 provides an overview of ECOA®: why it was developed, the general development approach and the benefits it offers.
- Section 7 provides a tour of key ECOA® concepts and terminology, placing these in context with each other.
- Section 8 introduces supporting concepts that should aid the understanding of how ECOA® is intended to be applied in practice.

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of complex real time software systems using a service oriented component based approach.

2 Warning

This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development should rely on the DefStan or BNAE publications of the ECOA standard.

3 Normative References

Architecture Specification Part 1	IAWG-ECOA-TR-001 / DGT 144474 Issue 6 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	IAWG-ECOA-TR-012 / DGT 144487 Issue 6 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	IAWG-ECOA-TR-007 / DGT 144482 Issue 6 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	IAWG-ECOA-TR-010 / DGT 144485 Issue 6 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	IAWG-ECOA-TR-008 / DGT 144483 Issue 6 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	IAWG-ECOA-TR-006 / DGT 144481 Issue 6 Architecture Specification Part 6 – ECOA® Logical Interface
Architecture Specification Part 7	IAWG-ECOA-TR-011 / DGT 144486 Issue 6 Architecture Specification Part 7 – Metamodel
Architecture Specification Part 8	IAWG-ECOA-TR-004 / DGT 144477 Issue 6 Architecture Specification Part 8 – C Language Binding
Architecture Specification Part 9	IAWG-ECOA-TR-005 / DGT 144478 Issue 6 Architecture Specification Part 9 – C++ Language Binding
Architecture Specification Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 6 Architecture Specification Part 10 – Ada Language Binding

ISO/IEC 8652:1995(E) with COR.1:2000	Ada95 Reference Manual Issue 1
ISO/IEC 9899:1999(E)	Programming Languages – C
ISO/IEC 14882:2003(E)	Programming Languages C++
SPARK_LRM	SPARK – The SPADE Ada Kernel (including RavenSPARK) Issue 7.3

4 Definitions

For the purpose of this standard, the definitions given in Architecture Specification Part 2 apply.

5 Abbreviations

API	Application Programming Interface
ASC	Application Software Component
COTS	Commercial Off-The-Shelf
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
ELI	ECOA® Logical Interface
ICT	Information and Communication Technology
IoC	Inversion-of-Control
IP	Internet Protocol
LRU	Line Replaceable Unit
OS	Operating System
QoS	Quality of Service
RTOS	Real-Time Operating System
PINFO	Persistent Information
SOA	Service-oriented Architecture
UML	Unified Modelling Language
VME	Versa Module Europa (bus)
XML	eXtensible Markup Language
XSD	XML Schema Definition

6 ECOA Overview

6.1 Background

Platform mission system software is expected to continue to grow in size and complexity due to increased reliance on software derived capabilities. This situation drives the need for improved software architectural approaches and these are the focus of ECOA.

There is a desire to reduce costs both in development and deployment of platforms while at the same time having the capabilities to be effective over the diverse array of theatres in which modern operations are conducted.

Future development programmes are likely to require more efficient collaborative software development as cost pressures increase and systems become complex. Effective collaborative working is expected to be an increasingly important factor in future contracts. In addition there is a desire to support an expanded software supplier base, driving innovation and reducing cost. All this is set in an environment of increasingly complex and connected capability.

The diversity of the platforms and sub-systems will require an architecture that can be applied to mixed safety-integrity and security-integrity systems. The architecture must allow the handling of faults in a robust manner in order to restrict fault propagation and mitigate any effects on the rest of the operational system.

A large part of platform development cost derives from the risk associated with integrating complex, software-intensive systems. Systems are continuing to grow in size and complexity and ECOA provides a way to manage this integration risk and help ensure that integration of independently developed subsystems is straightforward.

To maximise re-use of software elements they must be portable and computing-platform independent. They should be designed in such a way as to anticipate the need for system evolution. The process of re-validation of a system following upgrades is supported by modularization and abstraction of its software elements such that re-use of existing validation evidence is facilitated. The outcome should be a reduction time and cost associated with upgrades.

6.1.1 ECOA and the need for Improved Software Architectural Principles

Traditionally platform-level software-intensive systems for military aircraft have been developed both “top-down”, to meet a set of user requirements, and “bottom-up”, to incorporate modified off-the-shelf subsystems with pre-conceived functionality and interfaces.

Much of the knowledge about interface peculiarities and of why subsystems behave in the specific way they do is entrenched with suppliers and so the approach has become self-sustaining.

Thus system design has been pragmatic rather than principled.

As a result of this approach the interactions between subsystems and the host system are commonly not solely defined by the fundamental abstract function(s) of the subsystem in question.

- Subsystems may include functionality unrelated to their prime purpose which ideally belongs elsewhere, but having been included for convenience at design time.
- Subsystems may replicate functionality explicitly performed elsewhere to simplify a design-time negotiation, causing potential ambiguity in system data.
- They may include adaptations to cope with the peculiarities of subsystem interfaces – especially relating to timing and the frame to frame coherence of related data items.

- They may include local work-arounds and compensations for the subtle details of sub-system or host-system behaviour.

This has resulted in platform-level systems with a number of undesirable properties:

- Their characteristics are obscure, and at best are understood by a small number of experts who have gained knowledge of the idiosyncrasies of the system through experience.
- They are brittle. That is small changes, which may be inevitable for reasons of evolving requirements or obsolescence, can result in significant fracturing of the system and disproportionate difficulties and expense to repair and accredit it.

Thus system integration, test and upgrade are difficult, expensive and risky.

Such pragmatically-engineered systems tend to be intricate and convoluted because of their development history. This is a form of complexity which results from the design approach, rather than the requirement.

The modular, service-oriented, open architecture approaches of modern ICT have been shown to produce robust solutions which are capable of rapid integration and expansion.

There is a pressing need for more adaptable and more affordable military avionics systems. It is in this context that the ECOA programme was launched. ECOA is a software architecture aimed at providing interoperability and portability at software interface level for real time systems. ECOA enables the construction of a service-oriented architecture using a model based approach, which provides the capability to formally specify an assembly of Application Software Components, as well as their interfaces and deployment onto computing platforms. It is a key enabler for creating a library of truly interoperable product lines (Application Software Components and computing platforms) from across the industry.

ECOA does not prescribe where functionality should reside within the architecture. ECOA has developed the architectural principles, specifications and supporting infrastructure which will allow the functionality of avionic systems to be realised from a set of subsystems designed to offer access to their fundamental capabilities in terms of clearly defined service calls. This promotes modularity, ease of integration, re-use and adaptability, each of which will impact significantly on the affordability of future military avionic systems.

6.1.2 Rationale for an Open Standard

In order to achieve more efficient development and upgrade of avionics systems, the community needs to work together using common standards. These standards must support the integration and re-use of existing products (including Commercial-Off-The-Shelf (COTS) software) alongside the integration of newly developed products.

This indicates the need for published standards defining a common exchange format and standard interfaces that can be used to develop modular architecture components that can be exchanged and reused use by the subscribed community.

The community includes the traditional aircraft primes and suppliers, but by creating an open standard it is a goal to create a more open market for avionic software. For example small and medium enterprises may be able to provide innovative new capabilities.

6.2 Aims of ECOA

ECOA aims to reduce development and through-life costs for new collaborative development programmes by reducing risk in the integration of complex mission systems, promoting software reuse and improving competition in the software supplier base.

ECOA has been developed for avionic mission systems software in its widest sense but it is anticipated that the concepts and standards of ECOA would apply equally well in the land and sea domains, and in contexts where mission capability spans all three.

An aim of ECOA is to support implementation of the majority of the non-critical (including real time) software of a mission system with open, agreed interfaces; the longer term aspiration is to support more critical and mixed integrity software. As far as managing more critical software is concerned, ECOA supports this by defining a High Integrity Ada language binding [Architecture Specification Part 11]. An ECOA implementation must be capable of interacting with legacy / system specific areas that may not be hosted on the ECOA architecture.

In summary, ECOA is intended to help achieve the following:

- Source code level portability of software applications across diverse target environments to protect against future obsolescence
- Re-use of software applications over time and across vehicle platforms or programmes
- Interchangeability of application components
- Collaborative development processes
- Ease of integration (risk reduction for new build and system upgrade)
- Scalability of application size and complexity (through model driven design and code generation)
- Scalability in terms of capacity and throughput (by deployment onto more capable hardware)
- Scalability in terms of multiply instantiating Components and Modules
- Scalability in terms of deployment of components across multiple computing platforms and/or vehicles
- Commonality (of application software across platforms)
- Ease of deployment & integration of application components
- Interoperability (between subsystems of different provenance)
- Usable in high assurance safety & security contexts
- Usable in systems with real-time behaviour
- Rapid technology insertion (through the use of well defined interfaces)
- Support an open market for application software development

In addition ECOA does not increase the effort involved in achieving the following:

- Configurability (behavioural variability in the implementation)
- Tolerance (to cope with mission modes and resilience to failures)
- Synchronised training (use of common components in live and training systems)
- Exportable (configurable for export)
- Catering for legacy applications (encompass and interact with legacy systems)
- Catering for integration of COTS and other non-ECOA software
- An architecture that supports the majority of a combat-air mission system

6.3 Approach to ECOA

ECOA seeks to exploit architectural concepts and systems engineering techniques that are already widespread in other industry sectors in the development of complex information systems.

ECOA is intended to be used by processes that produce:

- Component-based software;
- Loosely-coupled, service-oriented architectures;

It is intended that ECOA should be applied using model driven development.

ECOA provides the ability to develop statically defined Publisher-Subscriber data-oriented architectures.

These approaches offer increased flexibility, but ECOA also recognizes the need for rigorous qualification and certification in the target avionics software environment.

ECOA allows for the:

- Deployment of ECOA technology on a variety of hardware and software platforms;
- Integration of ECOA applications with non-ECOA applications.

6.4 Objectives for an ECOA conformant system

ECOA helps its adopters to achieve the stated aims by specifying the following:

In relation to Process

- Aspects of the software development process that supports modelling at a platform independent level, deployment into a platform specific form and implementation in a predetermined manner.
- Tools to support the full ECOA lifecycle, including those that might be held by third parties.
- Tool support for the generation of integration code to support deployment of components on a given software platform.

At a Platform Independent level

- Application components, in a rigorous, machine readable manner that includes what is provided, what is required in support, and aspects of "quality of service" (required and provided). Using this, the suitability of a component for the intended purpose may be assessed.
- Assembly of components in a logical configuration, which is independent of any physical computing environment.

At the Platform Specific level

- Deployment of components across protection domains / computing nodes in an integrator-defined configuration.
- Execution of application components in accordance with the ECOA standard, including triggering of operations (functions, procedures) from external events, mechanisms to control sequencing (using threading & queuing), and synchronisation.
- Interactions between components (local or remote) defined in terms of the ECOA prescribed mechanisms, specified in a more formal manner (e.g. using XML).
- System management according to a prescribed paradigm (to be consistent across a platform at least), which should include Initialisation, (re-)configuration, Health Monitoring, Fault Management.

At the Implementation level

- The building of software components together with the necessary integration code (containers) to form the linkage between components and the underlying software platform.
- Support for interactions between application components by specified Component APIs (request-response, events, versioned data).
- Generic APIs for accessing infrastructure services provided as appropriate for the context of usage (generic infrastructure where possible), which should include time, logging, error handling, persistent information access.

Concerning Safety & Security

- The definition of a High Integrity Ada binding
- The ability to capture temporal properties using Quality of Service attributes

Additional safety and security requirements (e.g. data integrity checks, authentication functions, determinism, level of assurance) may be specified as additional platform procurement requirements [Architecture Specification Part 5]

7 Key ECOA Concepts

In this section, wherever key concepts and elements of ECOA terminology are introduced these are highlighted in ***Bold Italic*** and, thereafter, Capitalised. The reader may refer to Architecture Specification Part 2 for clarification of terms.

Figure 1 is a diagram summarising the main artefacts of ECOA. Aimed at those less familiar with ECOA, the figure and its accompanying text provide a contextual framework within which to position more refined details that may be encountered later.

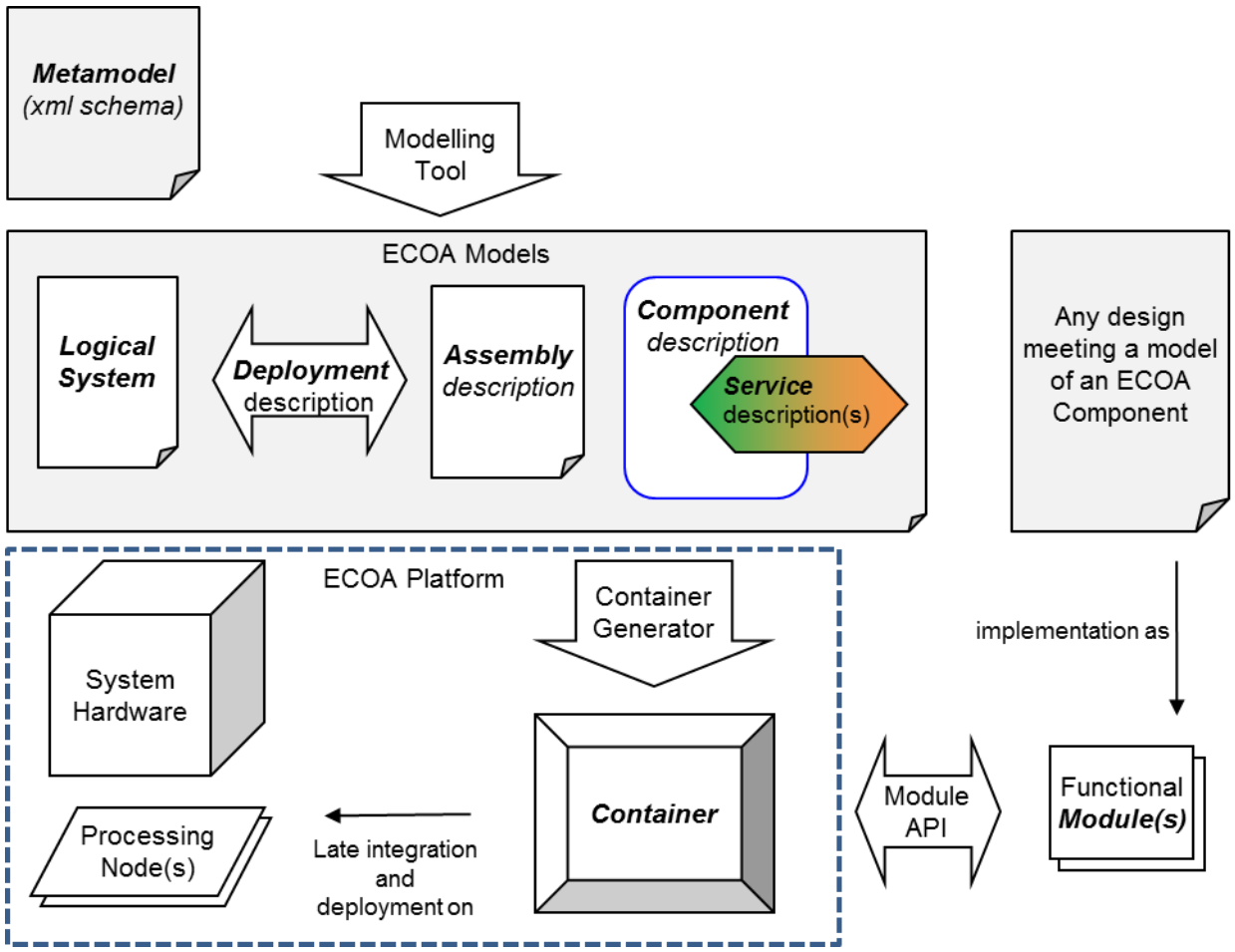

Figure 1 Introductory Overview of ECOA Artefacts

The ***ECOA Metamodel*** specifies how to model system descriptions/designs. It is a rigorous specification of the content of an ECOA model, and is part of this ECOA standard. ECOA models in XML that are compliant with this Metamodel are exchangeable between ECOA Platform toolsets.

An ***ECOA Platform*** furnishes the capability to deploy and execute ECOA models on specific types of target hardware and software infrastructure (if any).

In ECOA models the system software is described and organised into a set of collaborating ***Components*** called an ***Assembly***. The Components collaborate by invoking each other's ***Operations***. The Operations are collected into groups called ***Services***.

The ECOA model also identifies how the component software will be deployed on the available computing resources (Processing Nodes and System Hardware) identified in the ***Logical System***.

It is intended that the information in the model is used to automatically create dedicated infrastructure software. The dedicated infrastructure software includes Containers that will furnish the interfaces needed to support the application developer's functional code.

The Container is generated from the model. Each application is thus provided with a different ECOA API, one tailored to satisfy its specific functional needs.

The Container is integrated with the ***ECOA Modules*** (functional modules that the application developer creates) late in the development lifecycle. This means that if either Container or Module changes, such as during redeployment or after functional change, there is less expenditure of effort re-establishing a complete system.

7.1 Application Software Components and Services

From the top-down perspective of overall system architecture, software design in ECOA is expressed in terms of ***Application Software Components*** (**ASC** or '**Component'**) and the ***Services*** that they provide to, and require of, each other.

In an ***ECOA System***, an Application Software Component embodies some element of system functionality, and will have well-specified behaviour (functional, temporal etc.) which may be tailored through Component ***Properties***. ***Insertion Policies*** that accompany an ASC Definition express those necessary characteristics of any platform that is to host an instance of such a Component. This is described from an abstract point of view in Figure 2.

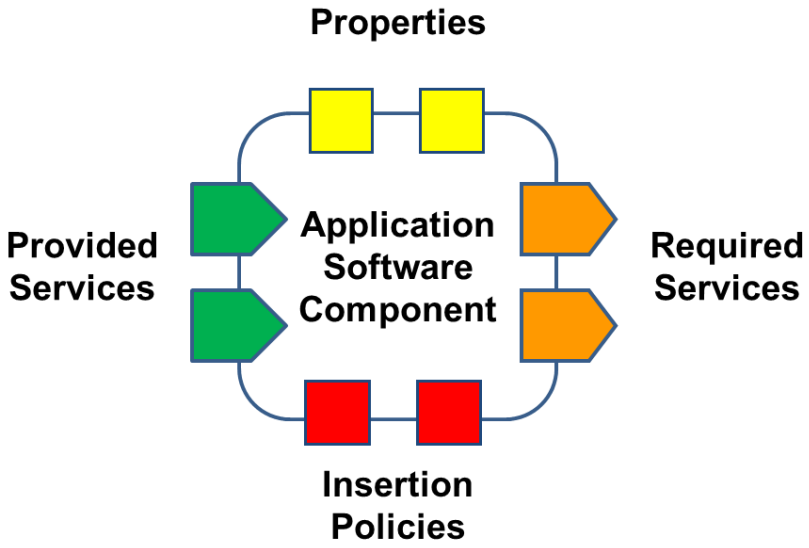

Figure 2 Abstract View of an Application Software Component

Figure 3, below, shows the relationship of entities that make up an ***Application Software Component Definition.*** Artefacts of this kind serve as formal specifications: contracts to which a Component developer will work in order to develop ***Application Software Component Implementations.***

For a given ASC Definition, a system integrator may be in a position to choose from very different ASC Implementations from different suppliers, but in terms of the Properties and Services they expose they will be indistinguishable, although they may be differentiated in the ***Quality of Service*** (***QoS***) they can provide.

A Service in ECOA comprises a group of ***Service Operations*** through which the Component that is the client (of a ***Required Service)*** may access the facilities of the ***Provided Service***. Service Operations may take the form of publish/subscribe ***Versioned Data*** access or various flavours of ***Event*** and ***Request-Response*** exchanges between connected Components.

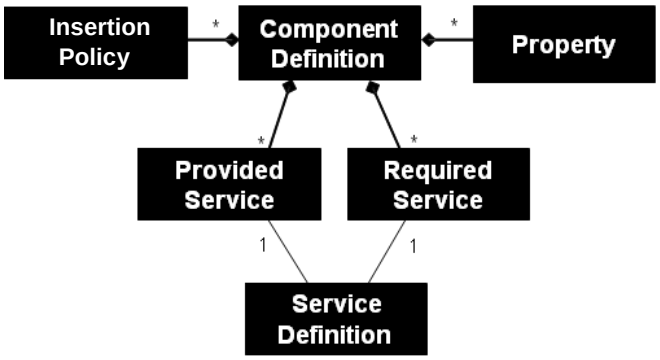

Figure 3 Simplified Entity-Relationship Diagram for a Component Definition

Architecture Specification Part 3 contains a full description of the Service Operations that can be used to make up a Service Definition

Architecture Specification Part 7 *provides the exhaustive specification of the ECOA metamodel where entities and cardinalities depicted in Figure 3 can be found.*

7.2 Architectural Design and the Assembly Schema

The architecture of an ECOA based system is defined from a Component and Service Oriented Architecture perspective.

A system's architecture is assembled by linking Components together according to SOA principles:

- in defining the interactions between ASCs, a requirer of a Service and the provider of that Service will refer to a common ***Service Definition***.
- Quality of Service attribute compatibility is taken into consideration when constructing ***Service Links***.

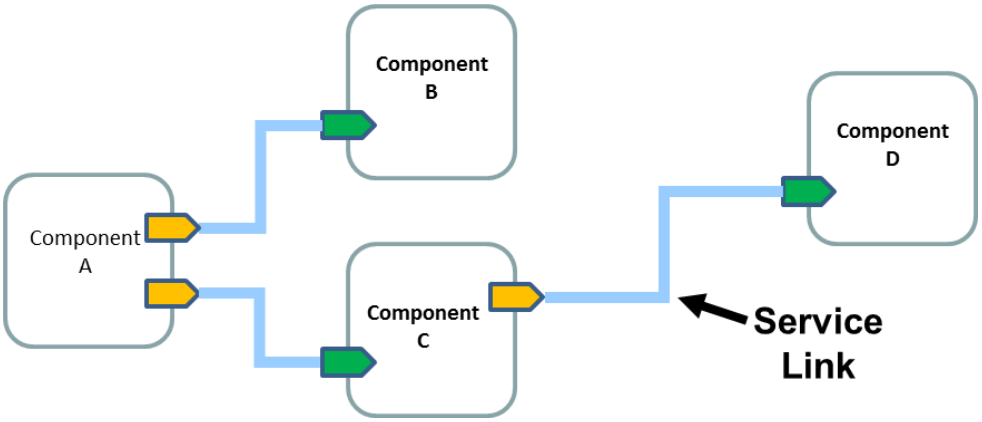

Figure 4 Simplified Representation of an Assembly Schema

An ***Assembly Schema*** comprises the set of declarative artefacts that are created during this design process. Figure 4 shows a diagrammatic representation of an Assembly Schema for a simple system comprising four Components and three Service Links.

A designer may decide to create new Service and Component Definitions, but will typically consider pre-existing Components and Services that could fulfil the system's functional requirements, and will model how these should be linked together. Though existing ASC Implementations may be differentiated in terms of QoS (which will always be implementation-specific) they may come with some particular Insertion Policies, this phase of design is broadly technology agnostic.

The final stages of design must accommodate the physical computing environment. A ***Deployment Schema*** is used to define how the executables within an ECOA Software System are to be distributed across its computing nodes.

7.3 ECOA XML Metamodel and Early Validation

An ECOA System is described using XML declarations that comply with the ***ECOA XML Metamodel***. Tool support for compliance checking versus this metamodel may help to ensure that the description of a system is internally self-consistent.

Architecture Specification Part 7 *provides the reference material that defines the ECOA XML Metamodel.*

The Assembly Schema works with other ECOA concepts to facilitate an ***Early Validation*** approach in which a system designer may gain confidence in a design, and to do so well in advance of final integration: i.e. that the system, once completed, will meet its functional and non-functional requirements.

For example, the provided and required Services, at each end of a Service Link, can be checked to ensure they have compatible QoS attributes. Analysis in this area may identify, at an early stage, parts of the design where timing or other issues are critical to correct performance.

A further element of a system description is the ***Logical System*** definition. It is a description of a system's computing hardware (**Computing Platforms**/**Computing Nodes** etc.) and physical connectivity of the final system. Using such information, further Early Validation work can take into consideration how Components are distributed.

Having a declarative, machine-readable system description opens up scope for sophisticated model-checking in support of Early Validation. In addition, the precise, declarative format is suitable for generation from a wide variety of system modelling tools. This allows engineers to use familiar methodologies (e.g. UML or SysML) to model and check a system which will then be realised in an ECOA compliant form.

7.3.1 XML Artefacts and Modularity

An overview of the XML files that follow the XML Schema Definitions (XSDs***)*** of the ECOA XML Metamodel is given in Table 1[1].

[1] Some terms in this table are described in later sections of this document.

Table 1 XML files used for system description

Data Type Definitions:	describe the data types used in Service Operations
Service Definitions:	describe the Services and their Service Operations
Component Definitions:	identify provided and required services, referring to their Service Definitions, alongside associated Quality-of-Service definitions. Refer to Figure 3
Component QoS Definitions:	specify non-functional characteristics for the Services relating to Components
Component Implementations:	define the executable entities, named Module Instances, that comprise a Component and the connections between them
Assembly Schema:	defines the ASC Instances and the service links between them
Logical System:	provides a model of a system's computing topology
Deployment Schema:	maps Module Instances onto the Logical System

As can be seen, much of the ECOA XML Metamodel is given over to aspects of ASC Definition, and ASC Implementation, with such declarations being partitioned into distinct artefacts.

This partitioning is designed to create a modular format which permits functional units to be independently specified and contributed by different parties for later integration.

Architecture Specification Part 7 *provides the ECOA XSD schema definitions.*

7.4 The Application Software Component Abstraction

A basic definition of an ASC is that it is a building block of a system. The power of the ASC abstraction lies in the different roles it fulfils:

a) Sections 7.1 and 7.2 described the modelling of an overall system architecture. From this top-down perspective ASCs are characterised by the Services they provide to, and require of, each other.

b) ASCs fulfil a key role in ECOA as the unit of exchange between software developers and/or integrators. This role is reflected in the focus on, and partitioning of, declarative artefacts pertaining to Application Software Components, as described in 7.3.1

c) Upcoming sections describe a bottom-up perspective in which ASC Implementations may be seen as aggregations of functional application code in the form of software Modules.

It is from these different perspectives that a system architect will approach the trade-off between logical top-down system decomposition involving the invention of new Service and ASC types, versus a bottom-up assembly process based on reuse of pre-existing Components and their Modules. An iterative architectural design approach may be called for.

ASC Definitions and other ECOA abstractions are expressed as declarations in XML at design-time. The nature of these declarations is preserved during the translation from XML into language and target-specific implementation code.

7.5 The Container and Inversion of Control

In ECOA it is the infrastructure code that controls the execution of the system specific code, provided by the Module Operations in response to the actions of other Modules. This is an ***Inversion of Control (IoC)*** with respect to traditional procedural programming in which application code commonly calls into the OS/Middleware to perform task scheduling activities.

The benefits of IoC are that it helps:

- To decouple the execution of a task from implementation;
- To focus a software implementation on the task for which it is designed;
- To provide the developer with contracts to be satisfied rather than concerns arising from how other subsystems are implemented;
- To reduce side effects when replacing software.

An implementation of Infrastructure code is called an ***ECOA Software Platform.*** It encompasses the ***Platform Integration Code,*** the computing facilities provided by the underlying operating system or middleware, as well as the means to interconnect with other ECOA Software Platforms.

Architecture Specification Part 5 *is the High Level Platform Requirements Reference Manual.*

An ECOA Software Platform provides, within its Platform Integration Code, ***Containers***: one for each or for several of the ASCs that it hosts. The Container and ASC are constrained to interact via their respective interfaces, which represent a set of custom, narrowed APIs which are designed to expose the minimum 'surface area' between an ASC and the ECOA Software Platform.

It is only through these APIs that an ASC may interact with the wider ECOA environment e.g. Infrastructure services and the Services of other Components. Scope for unwanted coupling is thus reduced and the prospect for future ASC reuse is enhanced.

The Container concept is an important one in ECOA. Containers are explained in greater detail in the following sections.

ECOA mandates ASCs to respect at least the following design principles:

- Inversion of Control, i.e. no control of their own execution (e.g. scheduling, threading).
- No use of legacy (e.g. OS and hardware) interfaces to communicate with legacy software or devices.

Only Driver Components, which are described in section 8.1.2, are allowed to bypass these design principles.

7.6 Software Modules

Software reuse is a motivating principle of ECOA. The concepts described up to this point support reuse at the Component and Service level. As section 7.4 points out, Components are system building blocks which will be assembled from (or decomposed into, depending on your perspective) smaller units – ***ECOA Modules*** (or simply Modules).

A Module embodies some functionality of a Component and ECOA seeks to impose minimal constraints on how its internals may be implemented. Modules are the unit of deployment in ECOA.

A declarative entity in the ECOA XML Metamodel called a ***Module Type*** provides the contract for implementing a Module, in terms of both the ***Module Operations*** it must implement and those that it depends upon. See Figure 5. Different ***Module Implementations*** may exist: perhaps targeting different hardware; perhaps written in different languages; but compliant with the same Module Type contract.

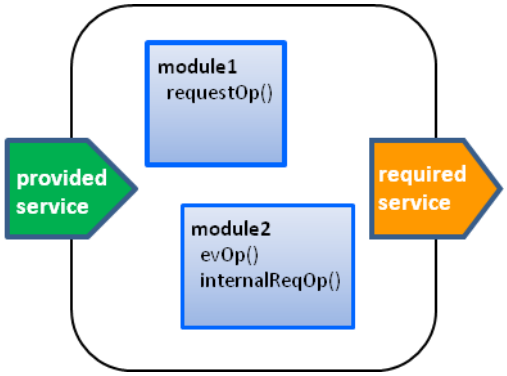

Figure 5 Modules comprising an ASC, and example Module Operations

Module developers can conceivably use any technology, languages, libraries etc. they need to get the job done. However, to guarantee portability and reusability a Module should operate by using only the facilities provided by the Container that surrounds it.

The precise specification for transformation of XML declarations to code means that portable Module Implementations can be developed: - Cross-compiled object code may be delivered to system integrators who can compose Modules from different suppliers together to form their systems, without the need to share source code or header files.

Following the Inversion-of-Control principle, ECOA Modules are passive and (with some specific exceptions detailed in 7.9) do not assume control over, or even knowledge of, the wider system, but instead are hosted and operated under control of the ECOA Software Platform.

Module Implementations should be re-entrant. Modules are instantiated by the ECOA Software Platform, and can expect the platform to invoke the ***Entry Points*** that implement Module Operations. A Container-provided thread is used for all interaction at a ***Module Instance's*** boundary with the ECOA Infrastructure: a Module Instance can only be executed by that one thread.

Where an ASC supplier requires internal state to be maintained in their Module Implementations, they shall use an optional ***User Context*** and an optional ***Warm Start Context***. When using these contexts, the ASC supplier can expect the platform to maintain them. The Warm Start Context can be used for warm restart of the Module. This allows functional data values to remain available after a warm restart recovery action is performed by the ECOA Fault Handler.

Note: the actual ECOA Platform capability of maintaining the Warm Start Context should be consistent with the recovery actions supported by the ECOA Platform.

Architecture Specification Part 4 *defines the Module User Context and the Module Warm Start Context.*

7.7 Module and Container Interfaces

Section 7.5 introduced the Container concept and the custom, narrowed API through which Containers and Components must interact. This API is realised at the ECOA Module level with the Container-facing aspect of a Module being termed the ***Module Interface*** and the Module-facing aspect of a Container termed the ***Container Interface***.

The Module Interface is derived from the Module Type. Module Interface entry points are handlers for Module Operation invocations coming via the Container from the wider ECOA System.

Methods exposed by the Container Interface provide the means by which Modules can call Module Operations on other Modules or Components, or make use of Infrastructure services. A Container Interface

implementation may be mechanistically derived (code generated) in its entirety from the Module Type, Module Implementation and Module Instance and supporting declarations.

Infrastructure facilities provided by the Container Interface include time services, logging and fault management. The lifecycle of Modules, Components and their Services are covered in greater detail in section 7.9.

7.8 Module Operation Links

Sections 7.4 c) and 7.6 both refer to the realisation of an Application Software Component Implementation from Module Implementations. An ASC Implementation may comprise many Module Instances, perhaps with multiple instances for a given Module Type. Such Modules Instances must be identified and connected together appropriately in order to fully elaborate the ASC Implementation.

Module Operation Links, depicted in **Figure 6**, define the connectivity among the Module Operations of a Component's Module Instances in addition to specifying which Module Operations are to fulfil an ASC Implementation's Services[2].

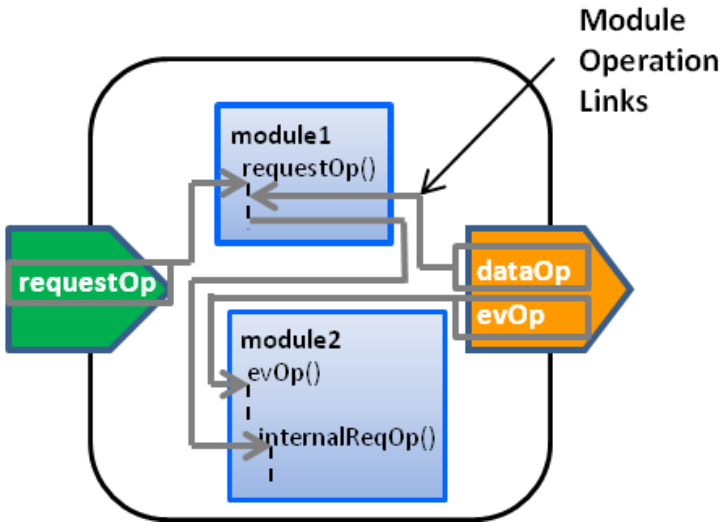

Figure 6 An Example Component Implementation

7.9 Control of System Functionality in ECOA

The subsections that follow describe standard ECOA approaches to the introduction of custom control of application execution.

Architecture Specification Part 4 *is the reference manual for these concepts.*

[2] Services are aggregations of Module Operations, just as ASC Implementations are aggregations of Module Instances.

7.9.1 Trigger Instance

Given the general IoC principal that Module Operations are entirely passive, the question arises as to how the developer should create active ASCs. A special pseudo Module called a ***Trigger Instance*** provides a mechanism to allow this.

A Trigger Instance is specified in XML and wired to other Modules like any Module Instance, but its implementation is provided by the Infrastructure. Using a Trigger Instance an ASC developer can implement periodic activation at an operation level without reference to the underlying OS/Middleware, and without the need to depend on activation from an incoming Request or Event from an external Module.

Figure 7 depicts the Modules Instances and Module Operation Links within a Component which provides no Services and is not stimulated by any external calls. It contains two functional Modules and a Trigger Instance which is responsible for triggering execution of the Component on a periodic basis.

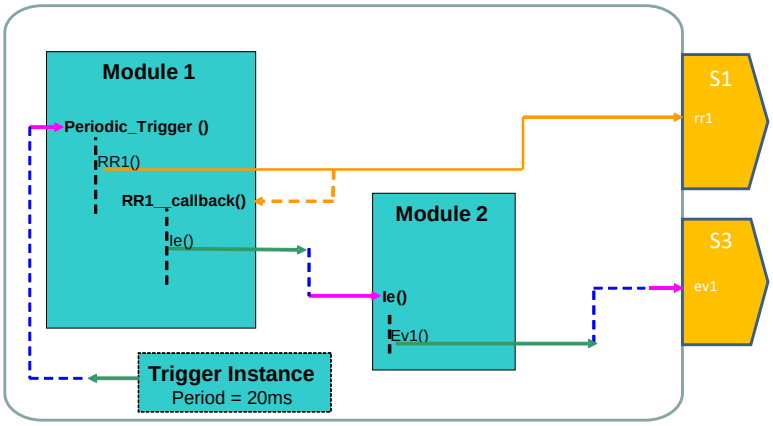

Figure 7 Trigger Instance linked to a Module within a Component

7.9.2 Module Lifecycle

The Infrastructure initialises, starts and stops the Modules Instances implementing the ASCs. The infrastructure achieves this by sending **Lifecycle Events** to each Module Instance.

When a Module Instance is not in the RUNNING state, any activation requests which are not lifecycle events are ignored.

The Infrastructure will generally control the lifecycle of Module Instances in response to higher level system events that affect the lifecycle of the ASCs as a whole (e.g. faults).

7.10 Hardware and Software Interoperability

The preceding sections describe the declarative entities and concepts required to define a Service and Component oriented system that may be executed in a real-time, embedded environment. A system thus defined is agnostic to any underlying technologies: programming language; middleware; (RT)OS; hardware; network etc.

To deploy and execute the system does, of course, require that the declarative entities and portable Module code must be targeted at a physical deployment environment.

Some of this targeting falls to the selected implementation of the ***ECOA Platform***: it will constrain, for instance: programming languages that may be used; types of middleware and families of (RT)OS that are supported.

7.10.1 Logical System definition and Deployment Platforms

ECOA defines the concept of a Logical System which is a logical definition of a computing infrastructure in terms of Computing Platforms, Computing Nodes, ***Protection Domains*** and ***Logical Links***. Protection Domains allow for spatial, and possibly, temporal isolation or partitioning in order to support multiple safety or security levels, for example. Logical Links are a simple abstraction of communication connections (e.g. VME or Ethernet) and are characterised by attributes such as bandwidth and latency.

Computing Nodes and Logical Links are characterised by simple attributes to enable modelling and assessment of a system prior to the completion of development (see Section 7.3 on Early Validation).

The definition of a Computing Node is deliberately abstract and may be a single core of multi-core processor, a single core processor or a multi-core processor.

An example of a Logical System is shown in Figure 8. It depicts machine1 connected to machine2 by a Logical Link. On machine1 there are 2 Protection Domains and on machine2 there is only one. On machine1 there must be a segregation mechanism at operating system level; whereas machine2 does not require this because it is only executing a single Protection Domain.

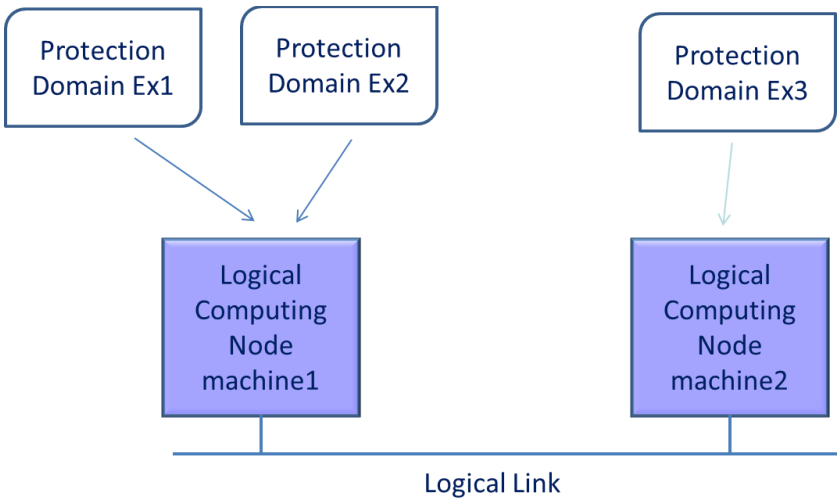

Figure 8 Example Logical System

The deployment of Components is described by a mapping of the Module Instances onto a Logical System. The description of this mapping is called a Deployment Schema and relates Module Instances, ***Container Instances***, Protection Domains, Computing Nodes and networks. This is shown in Figure 9.

One or more Module Instances are allocated to one Container Instance: the executable is the binary image containing the Container Instance and the Module Instances within a Protection Domain.

The possibility to deploy the Module Instances of an ASC Instance into multiple Protection Domains is a Platform procurement option.

One or more Protection Domains are allocated to any given Computing Node and communicate with other instances through an OS/middleware using physical links. A single instance of an ECOA layered software architecture executing on a single processing resource is termed an ***ECOA Stack***.

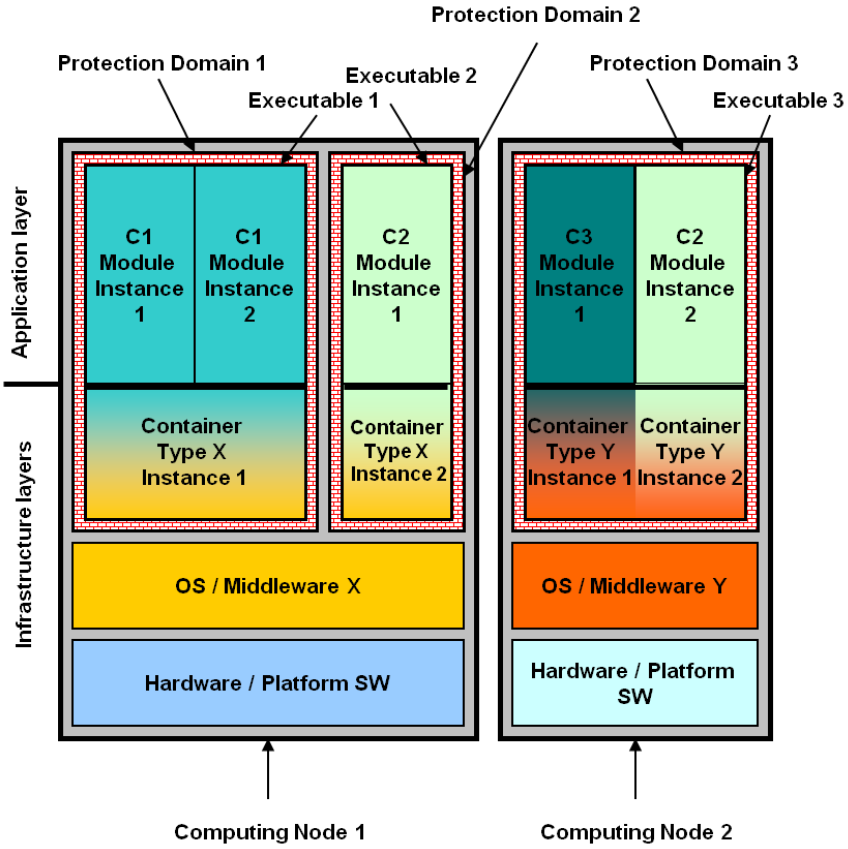

Figure 9 Deployment View

7.10.2 Interoperability Protocol: The ECOA Logical Interface

Communications between ECOA Platforms is achieved through the use of the ***ECOA Logical Interface*** (***ELI***). This interface is a standard, well defined protocol that is independent of the underlying physical transport media. Use of the ELI ensures that independently developed ECOA Systems or ECOA Stacks are able to make use of each-other's Services.

Service Definitions are composed of Service Operations with associated typed data and parameters. The specification of the data types is well-defined, allowing off-line checking of the model as well as on-line checks in languages that support this. The byte level payload for external communication of typed data over the ELI is precisely defined (in terms of endianness etc.)

All ECOA types exist within namespaces that can be nested. The following data type declarations are supported:

- *Predefined types* which are a set of basic types (e.g. uint32)
- *Simple types* which are a refinement of a predefined type or a simple-type itself to give a functional meaning to the type (e.g. list_index_type). Simple types can define bounds.

- Enumerations
- *Fixed records* containing fields of any other type
- *Variant records* that allow optional fields
- Fixed-size arrays
- Variable-size arrays

Component Containers use the facilities provided by the underlying operating system and/or middleware to provide the transport mechanism for the ECOA Logical Interface. For example, this could be via an Internet Protocol (IP) sockets mechanism using Ethernet or over a VME backplane. ECOA does not standardise the transport mechanism as there are a large number of mechanisms and the ways in which they can be used.

Depending on the transport mechanism employed, use of the ELI implies some overhead associated with uniformly representing data for communication.

Support of the ELI by a given ECOA Platform is a procurement decision. Thereby, it is possible to build and to procure ECOA Platforms that do not implement the ELI. This is aimed at target systems where it is a design assumption that ECOA ASCs will be deployed onto a single ECOA Platform. Though the ELI may be used internally by ECOA Platforms, implementations may take advantage of circumstances in which the implied overhead can be optimised away. For instance: given Modules of the same language, compiled with the same compiler and which are integrated into a single Protection Domain, then the ECOA API calling conventions and language bindings will ensure that the Modules can exchange messages using inter-thread communication without any intervening communications or ELI overheads.

7.11 Development Process and Tool Support

The ECOA XML Metamodel specifies the artefacts that are used for the exchanging of design information (refer to section 7.3).

It is an intention of ECOA:

- that it can be supported by a "model-driven" approach,
  - including exchange of model fragments
- that it supports progressive validation from an early stage in the development lifecycle,
- that it supports, as far as possible, an automated transition to implementation.

Figure 10 shows the general flow of the Component development/integration process and associated roles, relating to the ECOA XML files described above. The arrows on the left and right show, respectively, partial views of the Component development and integration processes.

The roles illustrated in Figure 10 can be summarized as follows:

- Vehicle Integrator:
  - Designs the system as an assembly of ECOA components (at system design level).
  - Responsible for specifying the components contracts and QoS, as well as the functional specification of the components.
  - Chooses a System Integrator, a Platform Supplier and Component Suppliers.
  - Performs the functional validation of the integrated system made of the component assembly on the target execution platform.
- System Integrator:
  - Specifies insertion policies of each component on the target platform so as to ensure that the intended system will be compliant with available physical resources.

- Performs the technical integration of the delivered ASCs.
- Fixes any issues in the component and platform specifications or any issues related to integration, during technical and functional integration phases.
- Is responsible for arbitration in case of modifications to be made in the system should there be any issues.

- Component Supplier:
  - Negotiates insertion policies with the System Integrator.
  - Develops and tests their component(s) according to the specifications of these components, their insertion policies and according to applicable development standards for these component(s).
  - Fixes any issues in the component reported during technical and functional integration phases.
- Platform Supplier:
  - Provides the target platform and related toolset.
  - Fixes any issues in the platform reported during technical and functional integration phases.

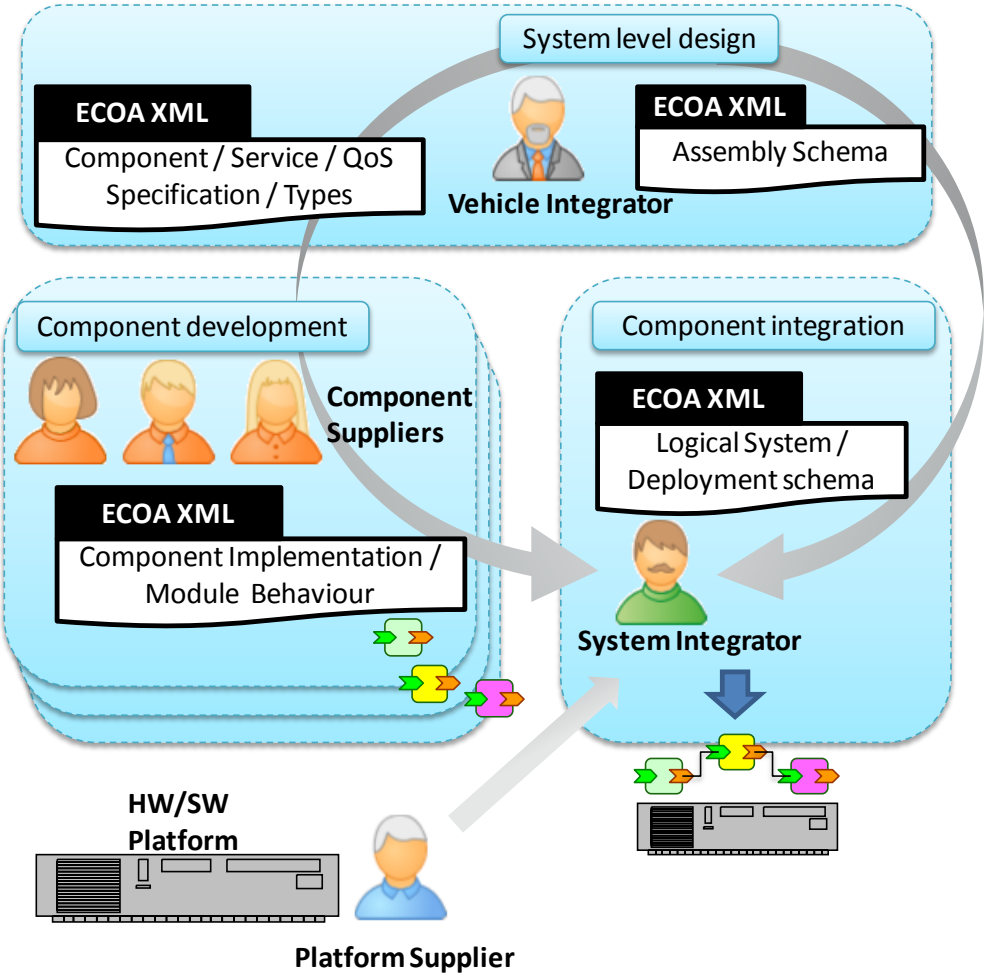

Figure 10 Component Development and Integration Process Overview

8 Supporting Concepts

8.1 Driver Components and Legacy subsystems

It is one of the key objectives to be able to deploy ECOA Application Software Components on non-ECOA legacy platforms and to be able to integrate non-ECOA software and hardware with an ECOA System. Figure 11 shows different cases involving integration of legacy subsystems which are discussed further, below.

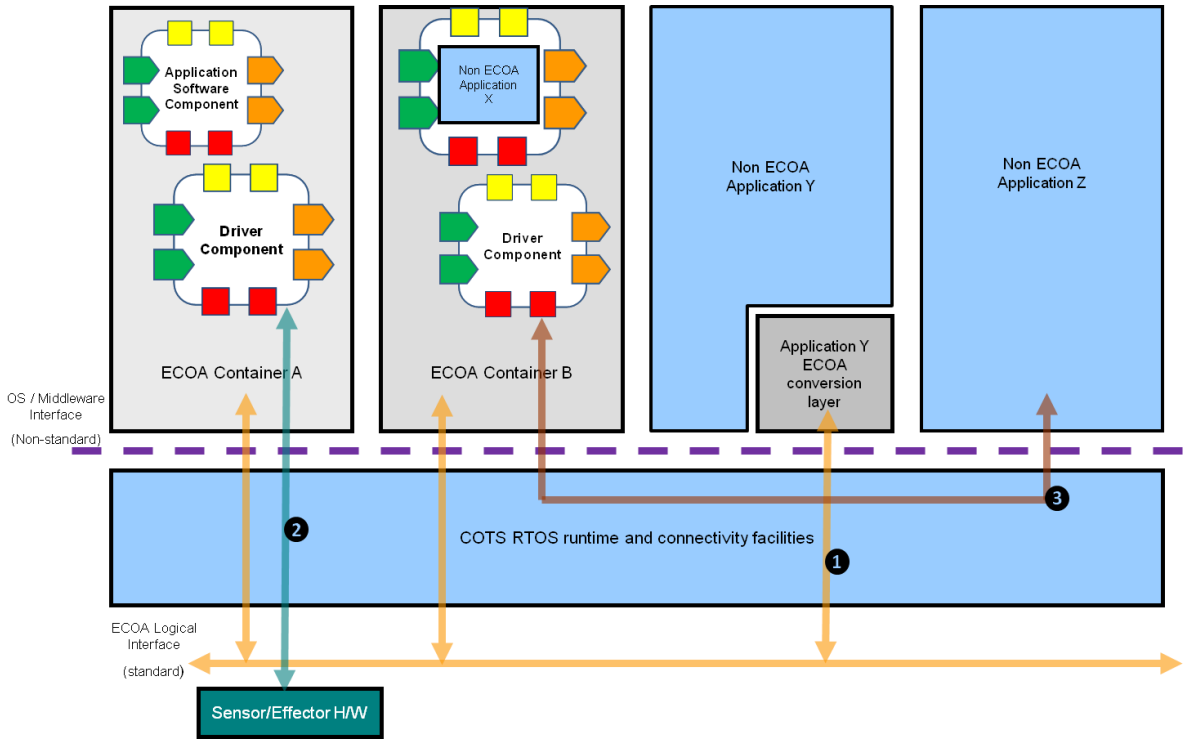

Figure 11 Integration of Legacy Software and Hardware into an ECOA Architecture

8.1.1 Legacy Software

The implementation of a legacy software application may not be consistent with the Inversion-of-Control principle. Legacy applications are likely to be closely coupled to an existing platform including operating system interfaces. Such applications may control their own execution (e.g. scheduling, threading), unlike an ECOA Application Software Component. This may imply the inability to effectively decompose application software into cohesive Modules, and may necessitate bespoke modifications.

A legacy system that consists of hardware and / or software can be integrated with an ECOA System using a number of methods including the following:

a) Wrapping or re-engineering as an ECOA Module that implements the necessary Inversion-of-Control behaviour (Application X in Figure 11)

b) Development of an ***ECOA Conversion Layer*** for a non-ECOA application to provide an interface compliant with the ECOA Logical Interface (ECOA Conversion layer embedded in Application Y in Figure 11)

c) Or, if the legacy application is (figuratively or literally) a “black box” then a dedicated Driver Component would be required. (Component of B connected to Application Z in Figure 11). See the next section on this topic.

8.1.2 Driver Components

The notion of a ***Driver Component*** is introduced to describe an Application Software Component that translates the interface protocol used by legacy hardware or software into operations specified in a Service Definition with well-specified behaviour. This is shown in Figure 11 : The Driver Component in Container A communicates with a non-ECOA sensor/effector via the connection marked ❷ and the Driver Component in Container B communicates with non-ECOA application Z via the connection marked ❸.

The software Modules that implement this kind of Component must behave as standard ECOA Modules in all interactions with their respective Containers (refer to 7.5), but they may, internally, use legacy (e.g. OS and hardware) interfaces to communicate with legacy devices. Such Driver Components will therefore be less portable than pure ECOA Application Software Components.

8.2 Component Reuse in Relation to System Architecture

Complex systems, such as avionic mission systems, require a structured organisation of their Components in order to be manageable. The ECOA programme has provided recommendations for a layered organisation of mission system Components, where the more generic Component Definitions reside in the lower layers and the vehicle platform-specific Component Definitions reside in the higher layers. This is illustrated in Figure 12.

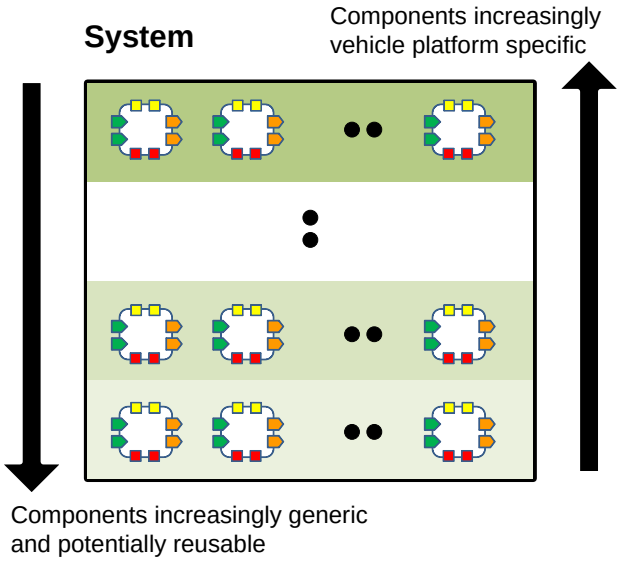

Figure 12 Layered / Hierarchical Component Based Architecture

Platform-specific Components typically embody top-level functional requirements specific to the platform. The potential reuse of these Components on other types of platforms is unlikely, whilst Components in the lower layers of the hierarchy are likely to be more generic and therefore the best candidates for reuse.

The same concepts hold *within* Component Implementations. The judicious separation of platform management functionality should enable the remaining Modules to be more generic, and hence better candidates for reuse within other Components.

8.3 Fault management

Fault Management is performed at various levels within the Infrastructure. The aim is to handle faults in such a way as to isolate, and minimize fault propagation between Components.

- Error Detection is the general term for detecting an error wherever it is detected
- Fault Handling is the general term for handling a fault wherever it is handled

Fault management involves a pattern made of:

- Detection of errors,
- Decision of recovery actions to be performed,
- Execution of decided recovery actions.

8.3.1 Error Categorization

Errors can be broken down into the following categories:

- Application Errors,
- Infrastructure Errors.

Thus, ECOA fault management is broken down into the following logical responsibilities, each of them applying the previously mentioned pattern:

- ECOA Fault Handling at application level (i.e. at Module level),
- ECOA Fault Handling at ECOA infrastructure level.

8.3.2 Key principles of ECOA fault management

ECOA fault management aims at handling Errors, in such a way that responsibilities are clear both within the ECOA system and between the ECOA system and the underlying execution platform, as shown in Figure 13.

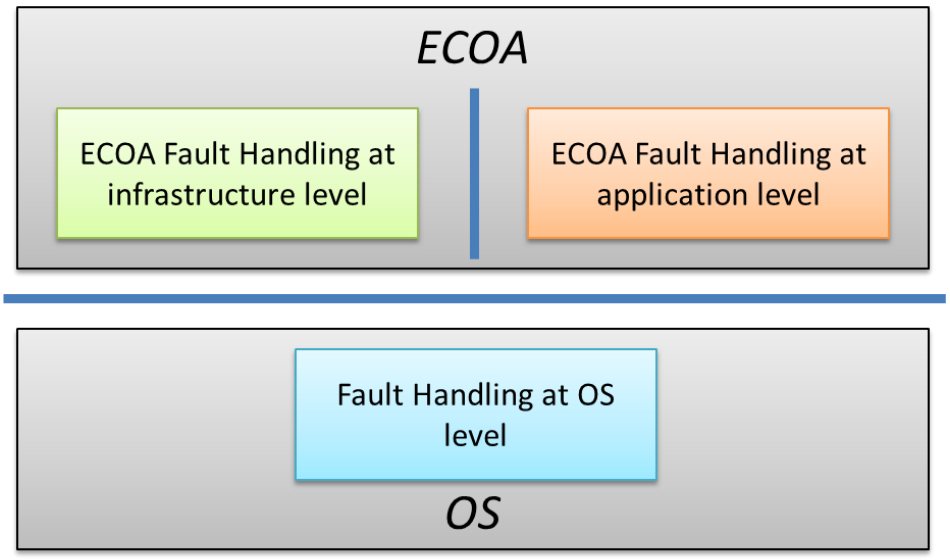

Figure 13 ECOA fault management responsibilities

In order to do so, ECOA fault management applies the following drivers:

ECOA Fault Handling at application level

- ECOA Fault Handling at application level focuses on applying potentially functional recovery actions for issues raised by ECOA modules.
- ECOA Fault Handling at application level can be broken down into two sub-levels, having in common the fact that they are of the responsibility of ASC providers:
  - The first level is being distributed into every application module across the ECOA system. Indeed, each module is responsible for taking into account functional issues that it detects at its level.
  - The second level of ECOA Error Handling could be the reason for implementing dedicated ECOA Fault Handling at ASCs level (i.e. at System Management level). This level is out of scope of the ECOA standard, as it is entirely dependent on the Mission System breakdown into ASCs.
- ECOA Fault Handling at application level and ECOA Fault Handling at infrastructure level may interact with one another under certain circumstances.

ECOA Fault Handling at infrastructure level

- Support of ECOA Fault Handling at infrastructure level by a given ECOA Platform is a procurement requirement decision.
- ECOA Fault Handling at infrastructure level is complementary with the underlying OS Fault Handling. Indeed, the purpose of ECOA Fault Handling at infrastructure level is to bring added value besides what the underlying OS might already offer as far as Fault Handling is concerned.
  - Consequently, for a given customer level of expectations regarding Fault Handling at system level, ECOA Fault Handling at infrastructure level capabilities may vary depending on the underlying OS capabilities. The more sophisticated the underlying OS is in that department, the simpler ECOA Fault Handling at infrastructure level can be (and vice-versa), as shown in Figure 14.

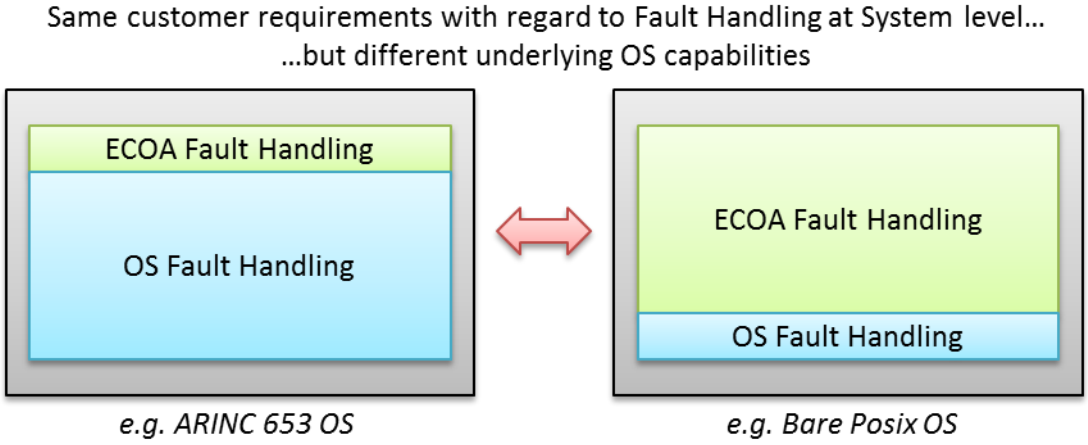

Figure 14 Complementarity between OS & ECOA Fault Handling at infrastructure level

- ECOA Fault Handling at infrastructure level may be implemented into one or several entities, called ECOA **Fault Handler**(s) on a single platform. The scope of the ECOA Fault Handler(s) cannot be wider than platform level (e.g. in a system made of two platforms connected through ELI, there shall be at least one ECOA Fault Handler entity per platform).
- ECOA Fault Handler(s) interface through the ECOA API.
- When being implemented as ASC(s), ECOA Fault Handler(s) shall be identified in the ECOA assembly (e.g. through an attribute in the metamodel).

- The system architect and system integrator are responsible for ECOA Fault Handling at infrastructure level.
- Given the previous principles, the ECOA standard aims at specifying the logical API between ECOA Fault Handling and the ECOA infrastructure, with respect to the following key drivers, as shown in Figure 15:
  - To define the list of Errors that may be received by ECOA Fault Handler(s), as well as the list of recovery actions that it may trigger in return.
  - With regard to the first principle above, each platform may only implement a subset of these lists in their API, depending on the underlying OS capabilities and depending on what the customer requirements are from System level point of view.

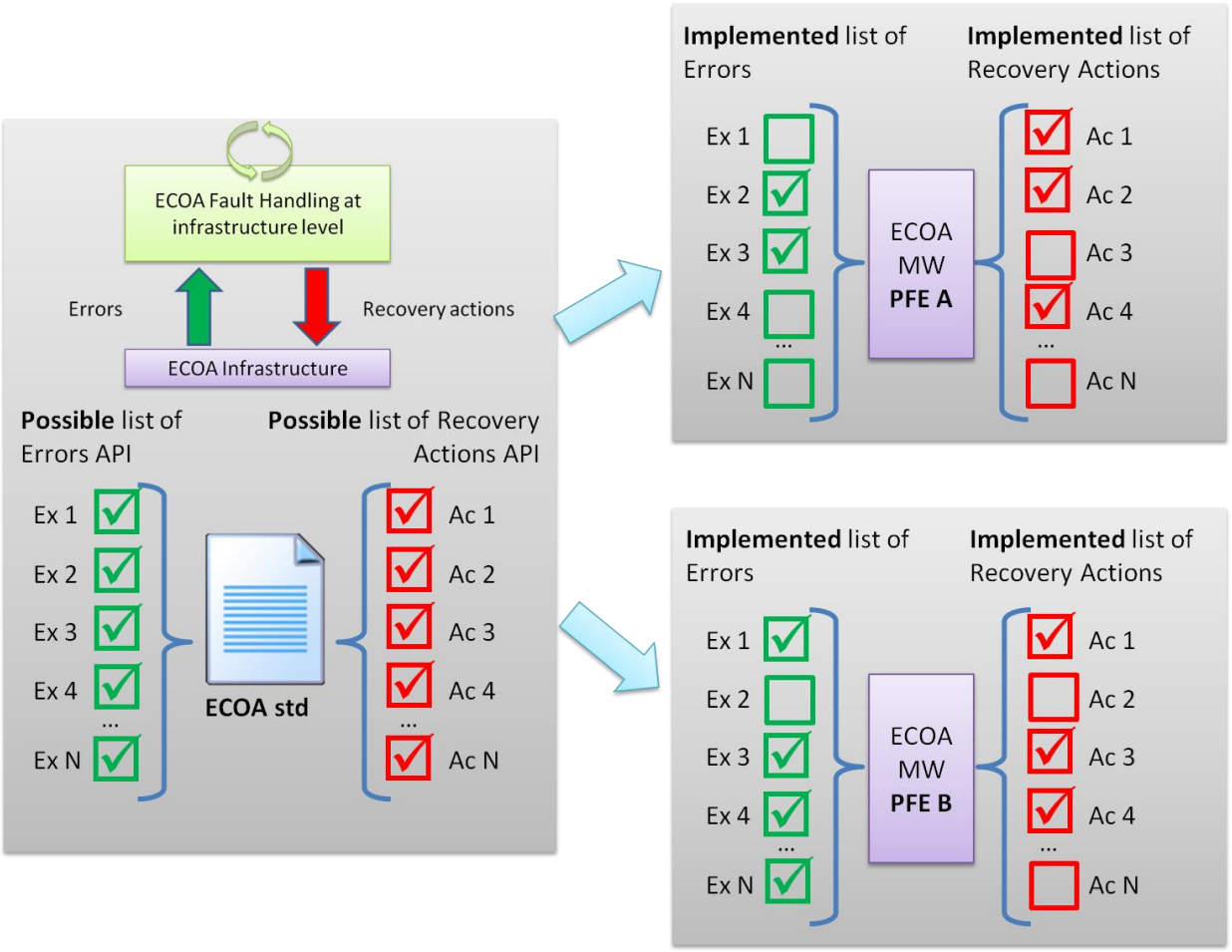

Figure 15 ECOA Fault Handling at infrastructure level logical API versatility Vs platforms

- Ref. Architecture Specification Part 4 provides the specification of ECOA Fault Handling at infrastructure level API.
- Depending on the platform, there may be different technical implementations of ECOA Fault Handling at infrastructure level as long as it complies with the logical API of the ECOA infrastructure. Technical implementations may vary in terms of architecture/usage (choice of the number of ECOA Fault Handler entities on the platform) and in terms of software implementation (e.g. it could be implemented as an ASC endowed with an extended API, or not).
- ECOA Fault Handling at infrastructure level does not address any functional issue. It does not have any knowledge of functional fault propagation chains throughout the system.

- This clear separation allows considering generic implementations of ECOA Fault Handling at infrastructure level. Indeed, its behaviour could entirely be configured by means of external datafiles (configuration tables), as shown in Figure 16.

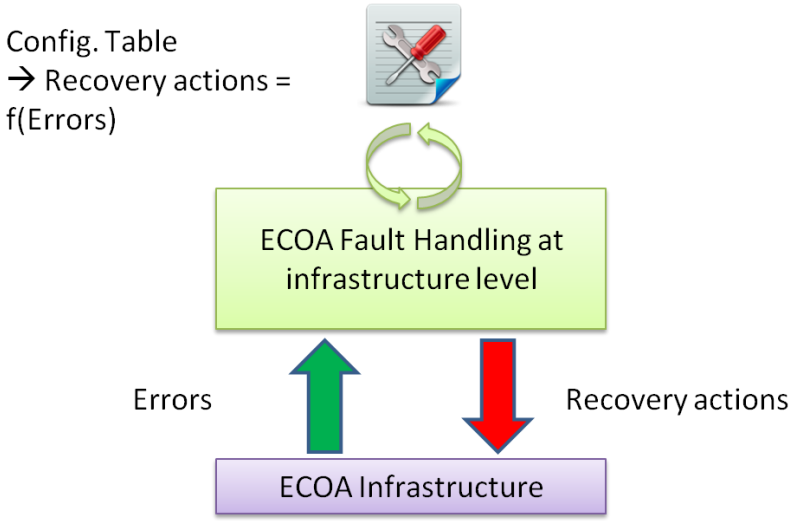

Figure 16 Illustration of a generic ECOA Fault Handler

Architecture Specification Part 3 *provides more detailed description of mechanisms related to ECOA fault management.*

8.4 Persistent Information

ECOA provides a way for ASCs to access persistent information, while remaining portable (i.e. agnostic to underlying platform mechanisms for importing and handling such persistent data).

“Persistent information” is data which will remain defined until it is explicitly deleted, even when the underlying platform is physically powered-off.

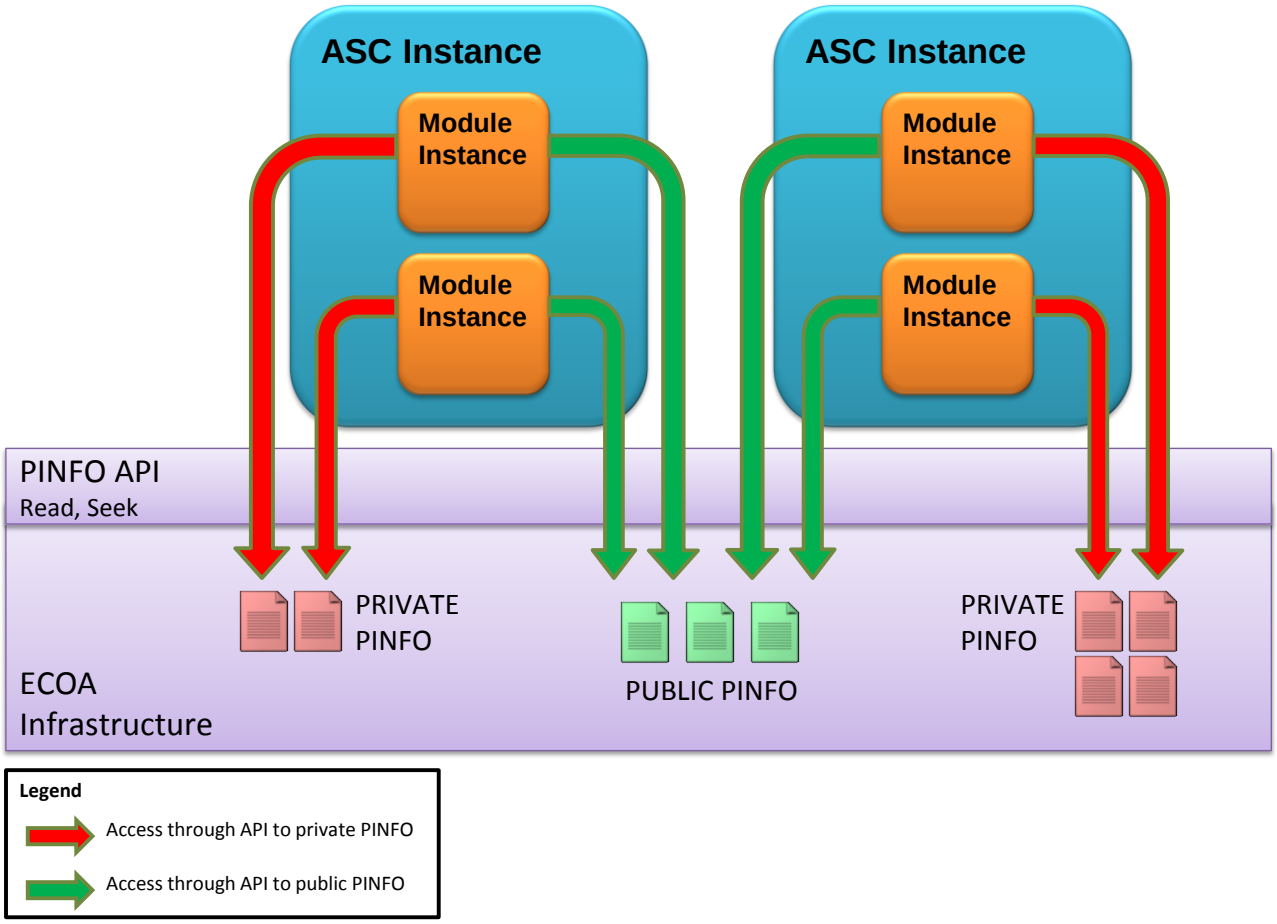

Figure 17 PINFO concept

ECOA provides an interface for ASCs to access PINFO through, read and seek operations, as illustrated in Figure 17. It is not possible for ASCs to access PINFO in write mode.

ECOA provides a way for declaring PINFO in an ECOA system and breaks them into two main categories:

- Public PINFO is data accessible by any Module Instance in the Assembly Schema.
- Private PINFO is data accessible by any number of Module Instances within the same ASC Instance.

Uses of PINFO could be:

- For an ASC to read private functional initialization data, such as sensor performance look up tables,
- For several ASCs to read public initialization data, such as public mission planning data.

NOTE: although providing some similar capabilities as a file system, PINFO is not as comprehensive or sophisticated.