European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification Volume I: Key Concepts

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

4 Abbreviations

API	Application Programming Interface
ARINC	Aeronautical Radio, Incorporated
ASAAC	Allied Standards Avionics Architecture Council
ASC	Application Software Component
COTS	Commercial Off-The-Shelf
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
IMA	Integrated Modular Avionics
IoC	Inversion-of-Control
IP	Internet Protocol
LRU	Line Replaceable Unit
OS	Operating System
QoS	Quality of Service
RTOS	Real-Time Operating System
SOA	Service-oriented Architecture
SW	Software
UML	Unified Modeling Language
VME	Versa Module Europa (bus)
XML	eXtensible Markup 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.

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 the improved software architectural approaches required to achieve this.

The standard is primarily focussed on supporting the mission system software of combat air platforms - both new build and legacy upgrades - however the ECOA solution is equally applicable to mission system software of land, sea and non-combat air platforms.

The ECOA specification is documented in four volumes, collectively identified as the Architecture Specification:

- Volume 1, this document, Key Concepts, introduces ECOA, its objectives, and the concepts and methods that are inherent to ECOA.
- Volume 2, the Developers Guide, covers the development of an ECOA system including software development, platform development and system integration.
- Volume 3, the Reference Manuals, specify technical details such as API language bindings and software interfaces.
- Volume 4 provides definitions for Common Terminology.

6 Architecture Specification Introduction

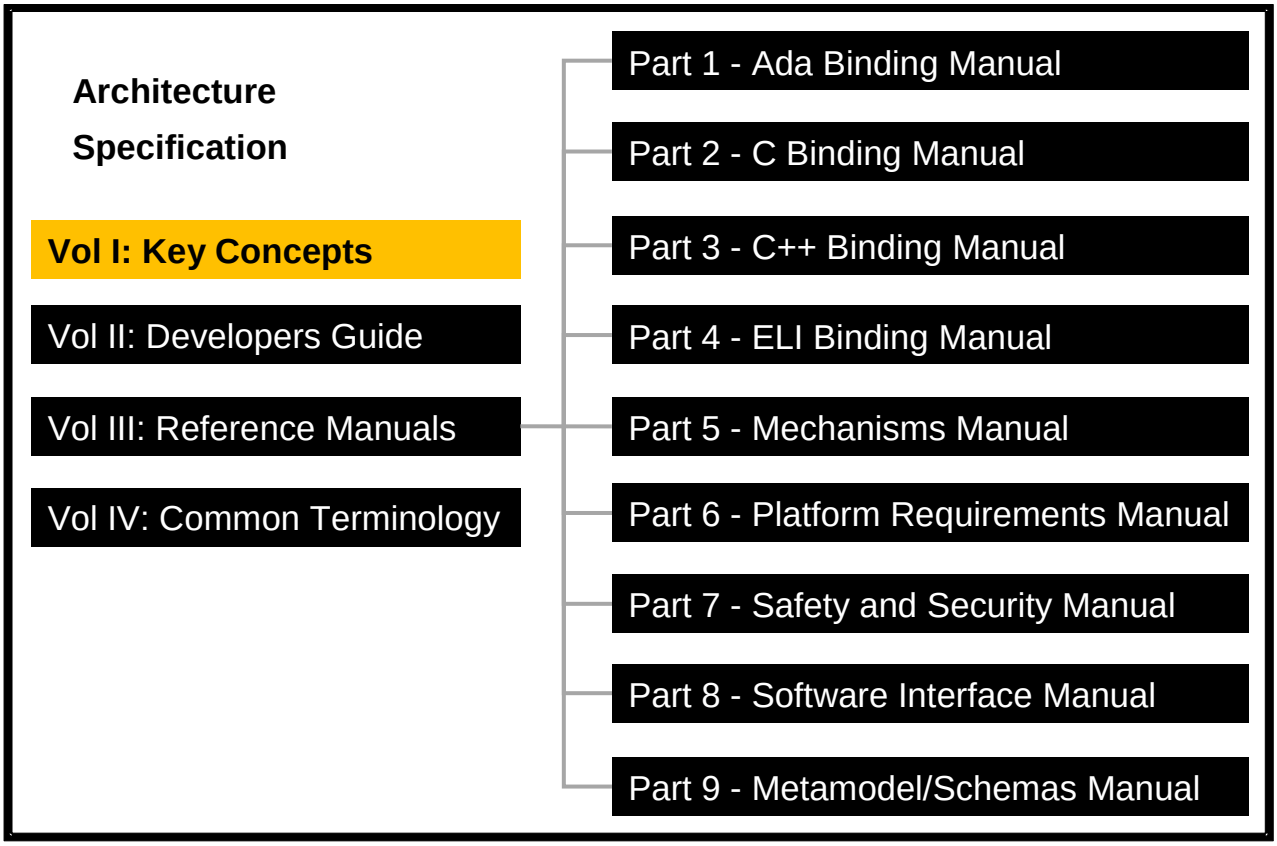

Figure 1 - ECOA Documentation

The Architecture Specification provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow developers to produce ECOA components and construct ECOA-based systems. It uses terms defined in the Common Terminology (Reference 11). For this reason, the reader should refer to this document, whilst reading this document. The details of the other documents comprising the rest of the Architecture Specification can be found in Section 10.

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

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

This document is Volume I of the ECOA Architecture Specification; it acts as an introduction to ECOA.

The document is structured as follows:

- Section 7 provides an overview of ECOA: why it was developed, the general development approach and the benefits it offers.
- Section 8 provides a tour of key ECOA concepts and terminology, placing these in context with each other.
- Section 9 introduces supporting concepts that should aid the understanding of how ECOA is intended to be applied in practice.

7 ECOA Overview

7.1 Background

In the future, platform mission system software is expected to grow in size and complexity. 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. Work share 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 support mixed safety-integrity and mixed security-integrity systems. The architecture will be required to handle 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 so the ECOA programme must provide 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 made possible. The outcome should be to the reduce time and cost associated with upgrades.

7.1.1 Rationale 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.

Such systems have been developed, or have evolved, without following a strong set of architectural principles governing the structure of the system or the interactions between elements of that system. Instead, system design solutions have been driven by what is available, or what adaptations can be negotiated with suppliers.

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 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 it.

Thus system integration, test and upgrade is 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, and the ECOA programme has set out to address that need.

ECOA is developing the architectural principles, specifications and supporting infra-structure 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 will promote modularity, ease of integration, re-use and adaptability, each of which will impact significantly on the affordability of future military avionic systems.

7.1.2 Rationale for An Open Standard

In order for all of the above to be achieved, it is necessary for the development community to work together using common standards. This standard 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.

7.2 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 is initially being 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 particularly in contexts where mission capability spans all three.

A prime aim is 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.

This aim requires that the underlying architecture is scalable from LRU to network level interaction and incorporates standard interfaces. It may require ad-hoc connectivity and distribution over multiple platforms supported by discovery, cooperation and interaction. The architecture will also be required to support real time service expectations of system components.

An aim of ECOA is to support implementation of the majority of the non-critical software of a mission system with open, agreed interfaces; the longer term aspiration is to support more critical and mixed integrity software. 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 achieve the following:

- Portability of software applications across diverse target environments
- Re-use of software applications over time and across platforms
- Interchangeability of application components
- Economical and collaborative development processes
- Ease of integration (risk reduction for new build and system upgrade)
- Scalability (of applications, in capacity, throughput, multiplicity)
- Commonality (of application software across platforms)
- Ease of deployment & integration of application components
- Rapid technology insertion (new solutions to the application needs)
- Configurability (behavioural variability in the implementation)
- Tolerance (to cope with mission modes and resilience to failures)
- Interoperability (between subsystems of different provenance)
- Multi-vehicle (deployment of components across multiple vehicles)
- Synchronised training (use of common components in live and training systems)
- Exportable (configurable for export)
- Usable in high assurance safety & security contexts
- Usable in systems with real-time behaviour
- Catering for legacy applications (encompass and interact with legacy systems)
- Catering for integration of COTS and other non-ECOA software
- Protection from future obsolescence
- Support an open market for application software development
- An architecture that supports 80% of a combat-air mission system

7.3 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. These include:

- Component-based software engineering;
- Loosely-coupled, service-oriented architectures;
- Model-driven engineering and model-driven development;
- 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 must also support:

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

7.4 Objectives for an ECOA conformant system

The ECOA programme proposes to achieve its 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 such as the ECOA Agency.
- 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 formalised manner that includes what is provided, what is required in support, and aspects of "quality of service" (required and provided): by which 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 a prescribed manner, 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 formalised manner (eg, 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 in 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, data).

- Infrastructure services provided as appropriate for the context of usage (generic infrastructure where possible), which should Include time, error handling, file system access.

Concerning Safety & Security

- Safety & security aspects "assured", which may include both functional (e.g. data integrity checks, authentication function) and non-functional (e.g. determinism, level of assurance) requirements / properties.

8 A Tour of Key ECOA Concepts

In this section, as key concepts and elements of ECOA terminology are introduced these are highlighted in ***Bold Italic*** and, thereafter, Capitalised. The reader may refer to Ref. 11 : Vol IV, Common Terminology for clarification of terms.

8.1 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.

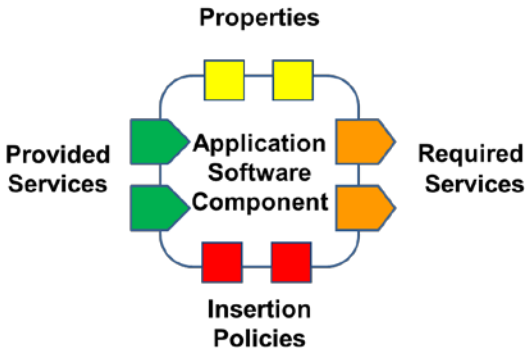

Figure 2 - An Application Software Component

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.***

In an ***ECOA System***, an Application Software Component embodies some element of system functionality, and will have well-specified behavior (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.

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 cohesive 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.

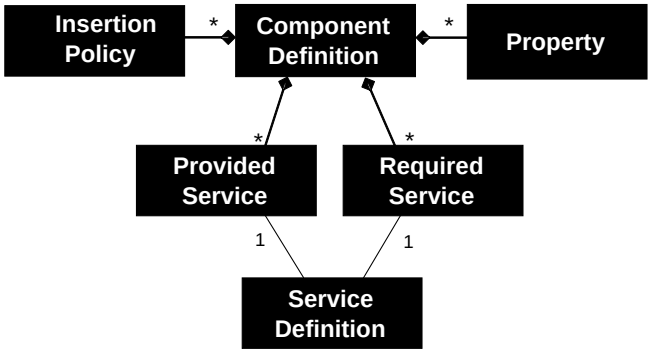

Figure 3 - Simplfied Entity-Relationship Diagram for a Component Definition

Ref. 6 contains a full description of the Service Operations that can be used to make up a Service Definition

8.2 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***.

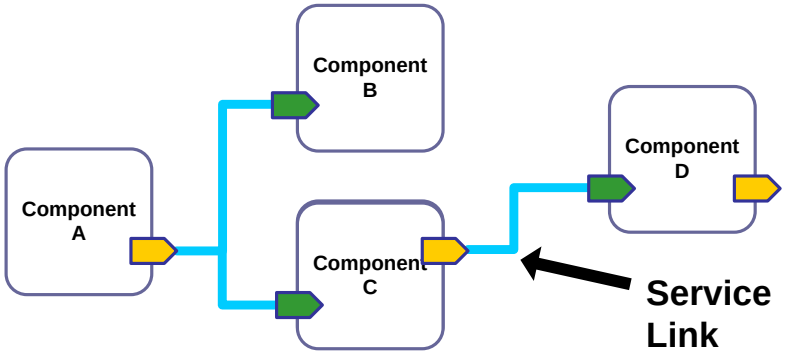

Figure 4 - Simplified Representation of an Assembly Schema

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) and 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.

Ref. 1 provides guidance on the process of software system design in ECOA.

8.3 ECOA XML Meta-model and Early Validation

An ECOA System is described using XML declarations that comply with the ***ECOA XML Meta-model***. Tool support for compliance checking versus this meta-model helps to ensure that the description of a system is internally self-consistent.

Ref. 10 provides the reference material that defines the ECOA XML Meta-model.

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 Nodes*** 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, to permit engineers to use familiar methodologies with UML or SysML etc. to model and check a system which will then be realised in an ECOA compliant form.

8.3.1 XML Artefacts and Modularity

An overview of the XML files that follow the XML Schema Definitions (XSDs***)*** of the ECOA XML Meta-model is given in Table 8-1 - XML files used for system description[1].

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

Table 8-1 - XML files used for system description

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

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

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: for storage in a local ***ECOA Library*** or for submission to an ***ECOA Agency*** for wider reuse.

Ref. 10 provides the ECOA XSD schema definitions..

8.4 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 8.1 and 8.2 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 8.3.1

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.

8.5 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.

Ref. 1 contains guidance for development of an ECOA Software Platform.
Ref. 7 is the Platform Requirements Reference Manual.

An ECOA Software Platform provides within its Platform Integration Code***, Containers*** : one for each or for several of the ASC Implementations 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 beyond: Infrastructure services and the Services of other Components. Scope for unwanted coupling is thus reduced and prospects 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.

8.6 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 8.4 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 Meta-model 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.

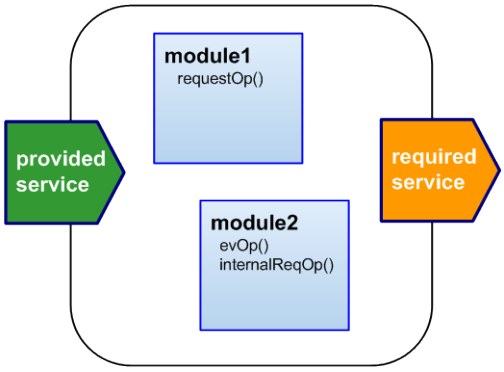

Figure 5 - Modules comprising an ASC, and example Module Operations

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.

Ref. 1 describes the process of Module software development and construction of ASC Implementations

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 8.9) 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 maintain a ***Context*** and to call into 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: one thread per Module Instance.

Ref. 9 defines the Module Context.

8.7 Module and Container Interfaces

Section 8.5 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. In addition, Container Interfaces for ***ECOA Supervision Modules*** provide methods through which a Supervision Module may manage the state of all Modules implementing the ASC. Supervision Modules and the lifecycle of Modules, Components and their Services are covered in greater detail in section 8.9.

8.8 Module Operation Links

Sections 8.4 c) and 8.6 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].

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

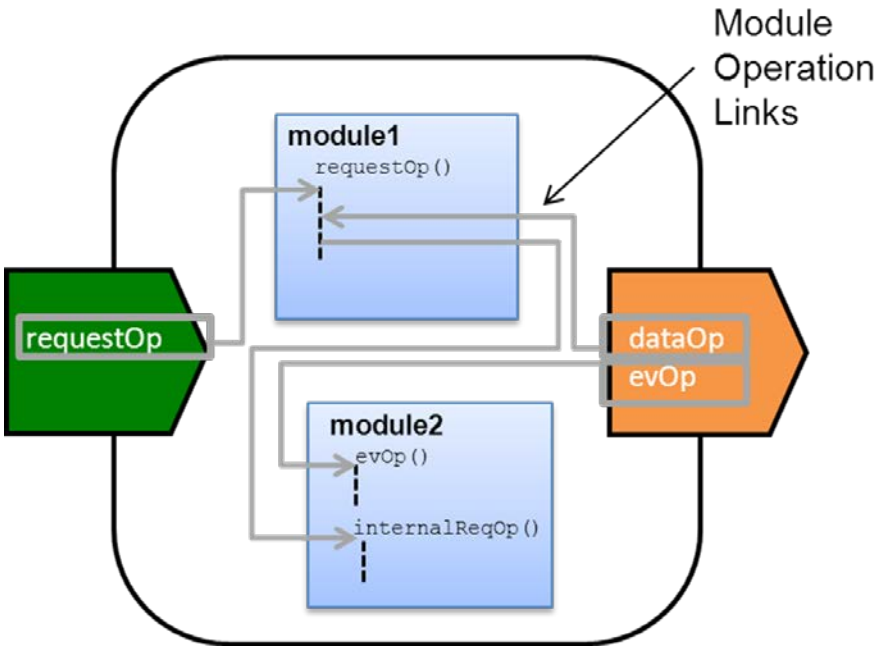

Figure 6 - An Example Component Implementation

Ref. 1 describes all aspects of the Module development process.

8.9 Control of System Functionality in ECOA

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

Ref. 9 is the reference manual for these concepts.

Ref. 1 provides guidance and rules for correct implementation of Component and Module lifecycle.

8.9.1 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 or 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, an ***ECOA Supervision Module*** and a Trigger Instance which is responsible for triggering execution of the Component on a periodic basis. ECOA Supervision Modules are described in 8.9.2

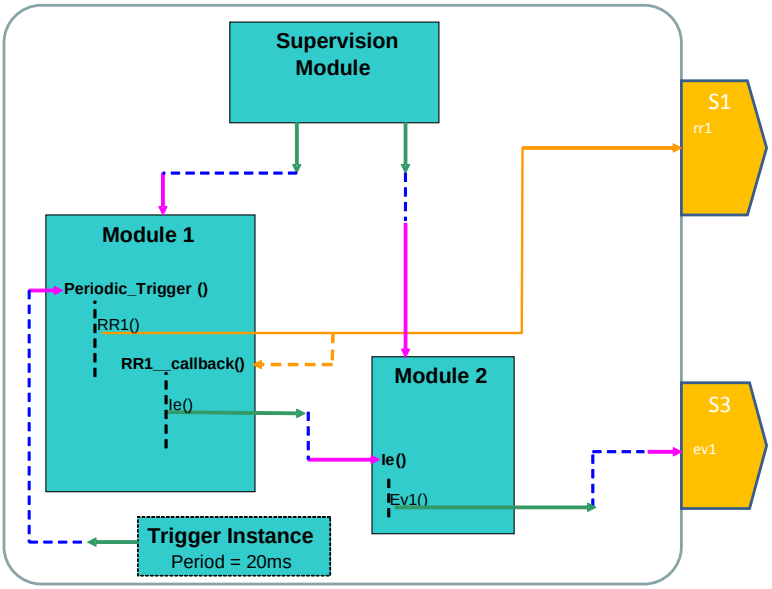

Figure 7 - Trigger Instance linked to a Module within a Component

8.9.2 ECOA Supervision Modules and Module Lifecycle

Figure 8, below, depicts the role of the ECOA Supervision Module as the part of a Component that implements Component-level concerns, such as ***Service Availability*** as well as implementing any ***Component Runtime Lifecycle Service*** (discussed further in 8.9.3).

The standard requires a Component Implementation to include only one Supervision Model.[3]

Supervision Modules are responsible for managing the lifecycle of the Module Instances that provide the functional implementation of an ASC.

A Component's Supervision Module is the only Module to be started by the Infrastructure and must explicitly initialise, start and stop the other Modules Instances implementing the ASC. It achieves this by acting through the Module Lifecycle API segment of its Container Interface.

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

A Supervision Module will generally control the lifecycle of Module Instances in response to higher level events that affect the lifecycle of the ASC as a whole, though a Component Implementation's design may result in such control being exerted for other functional or technical reasons.

8.9.3 Component Level Lifecycle and Functional Manager Components

At the Component level, the implementation of a ***Component Runtime Lifecycle Service*** allows one Application Software Component to monitor, control and manage the lifecycle of another Component.

Management logic at this level is likely to relate to functional requirements and could be compared to application-level management provided by IMA such as ARINC 653 or ASAAC, rather than system-level management. It is achieved, as shown in Figure 8, by one Component providing the Component Runtime Lifecycle Service and one Component requiring it. As such,

[3] A forthcoming issue of the standard may include provision for having default supervision behavior implemented as part of the Insfrastructure, allowing a simple Component to dispense with any Supervision Module

this is under the control of the system integrator and securely configured offline. As an example of a Component Runtime Lifecycle Service Operation, one Component can request that another Component starts or stops.

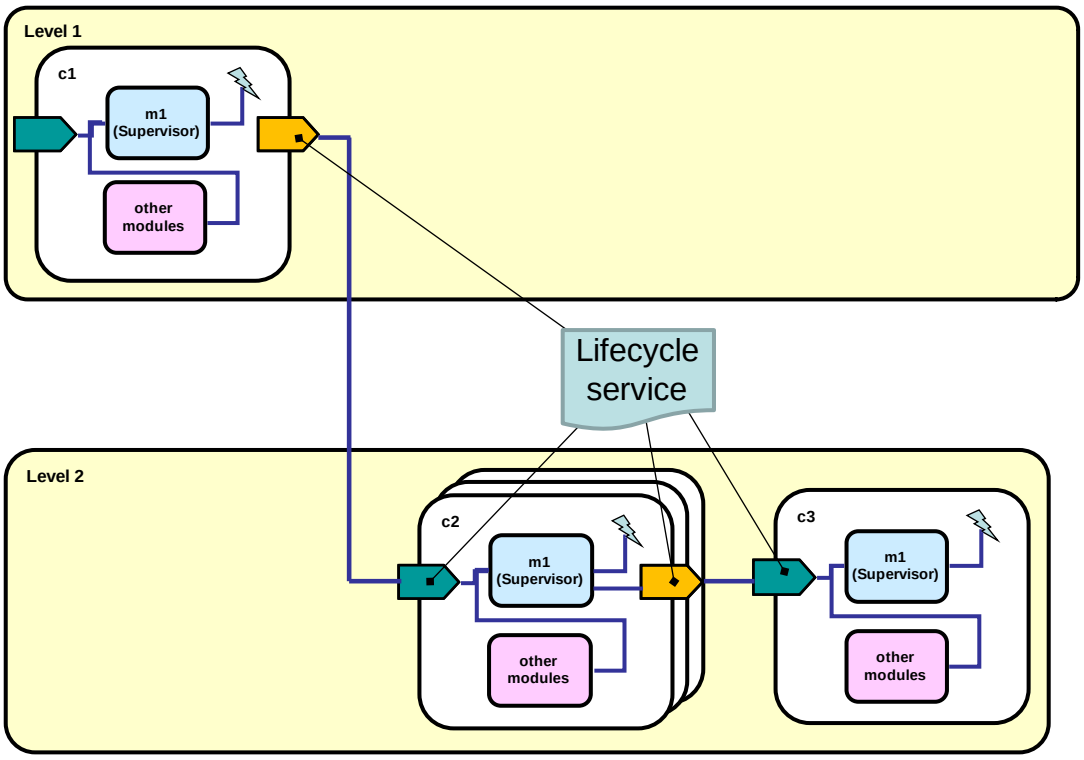

Figure 8 - Component Runtime Lifecycle Service

This scheme offers flexibility because it is under the control of the system integrator. Support for hierarchical system management is possible, reflecting the complex control needed in some systems. It is also possible for the start-up and shutdown of different Components to be completely asynchronous, reflecting a more loosely-coupled approach.

In addition to the Component Runtime Lifecycle Service, extra services may be used to implement functional modes which are meaningful when Application Software Components are running.

8.10 Hardware and Software Interoperability

The preceding sections describe the declarative entities and concepts required to define a Service and Component oriented system to be executed in a realtime, 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.

8.10.1 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 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 (eg. 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 8.3 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 below. 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.

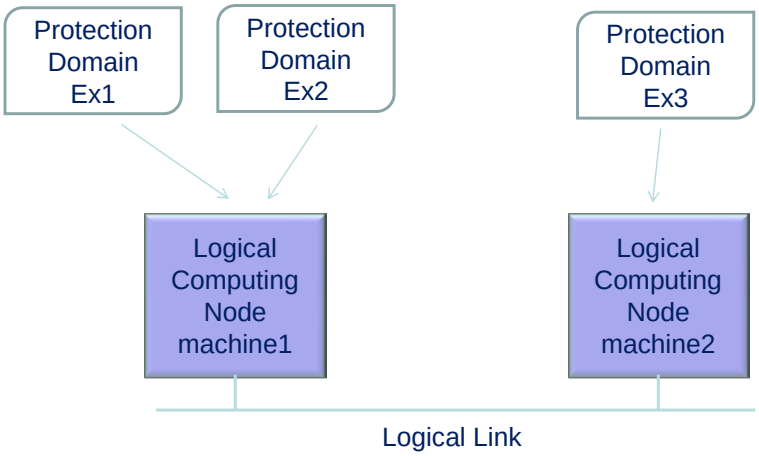

Figure 9 - Example Logical System

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 10.

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.

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***.

Much of the internal architecture of a Container is left to the ECOA Platform supplier, as there are many options. For example, if multiple Module Instances of an Application Software Component are mapped onto a single multi-core processor, options include:

- Allocating Modules statically to cores at build-time

- Dynamically dispatching Modules to cores at run-time[4]

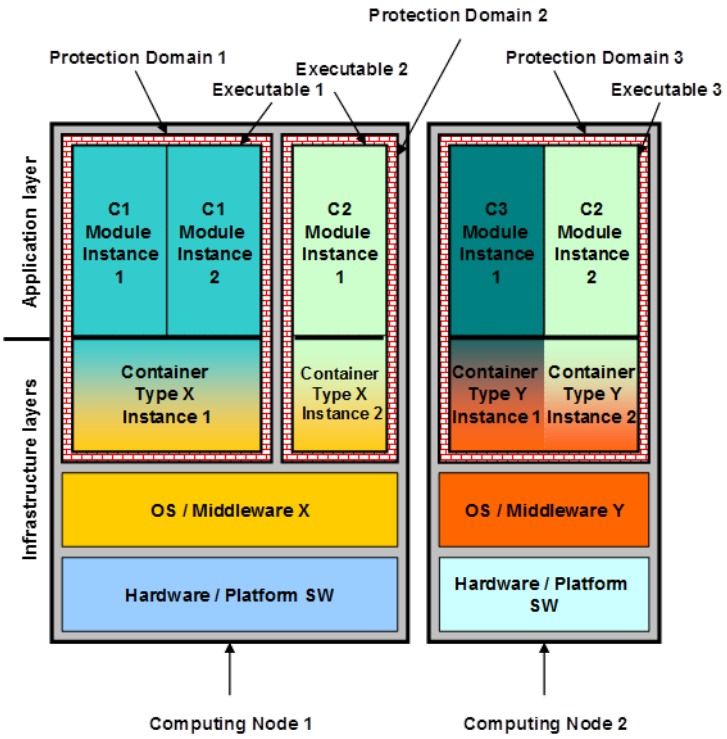

Figure 10 - Deployment View

8.10.2 Interoperability Protocol: The ECOA Logical Interface

Communications between Computing Nodes, and between the Container Instances that they host, 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 "wire format" for 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 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

[4] This concept is immature at the current issue of the Architecture Specification

- *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.

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

Though support for the ELI is mandatory for 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 simple inter-thread communication without any intervening communications or ELI overheads.

8.11 Development Process and Tool Support

The ECOA XML Meta-model specifies the artefacts that are used for the exchanging of design information (refer to section 8.3).

The ECOA XML Meta-model format, while precise and fit for purpose, is not suited for the capturing and modelling of system designs, and designers familiar with system modeling tools and methodologies would not wish to manually transcribe design models from tool formats to ECOA XML Meta-model format. It is anticipated that generic design, validation and transformation tools and plugins would be developed and provided by independent tool developers to form part of an ***ECOA Toolset*** that would assist the work of correctly generating the XML artefacts.

It is an intention of ECOA:

- that it can be supported by a "model-driven" approach,
- 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.

An ECOA Toolset would not be complete without a means of generating Container source code from XML and compiling this into object code – an unmanageable task if performed manually. ECOA Platform suppliers would be expected to include basic tool support for these development activities.

Figure 11 shows the general flow of the Component development and integration process, 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.

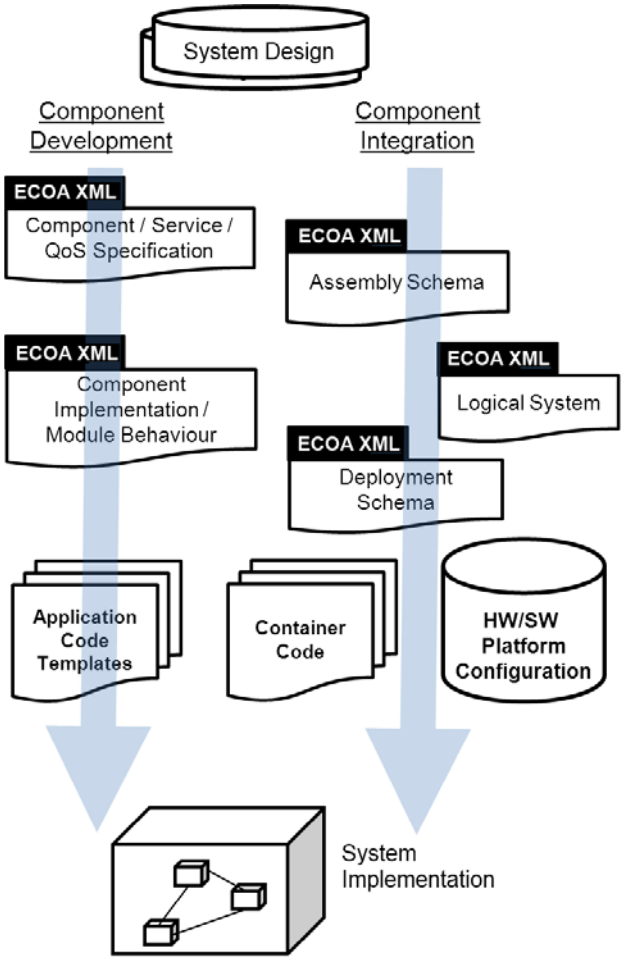

Figure 11 - Component Development and Integration Process Overview

9 Supporting Concepts

This section and its subsections summarise the main supporting concepts, which relate to practical concerns when designing a system using ECOA.

9.1 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 12 shows different cases involving integration of legacy subsystems which are discussed further, below

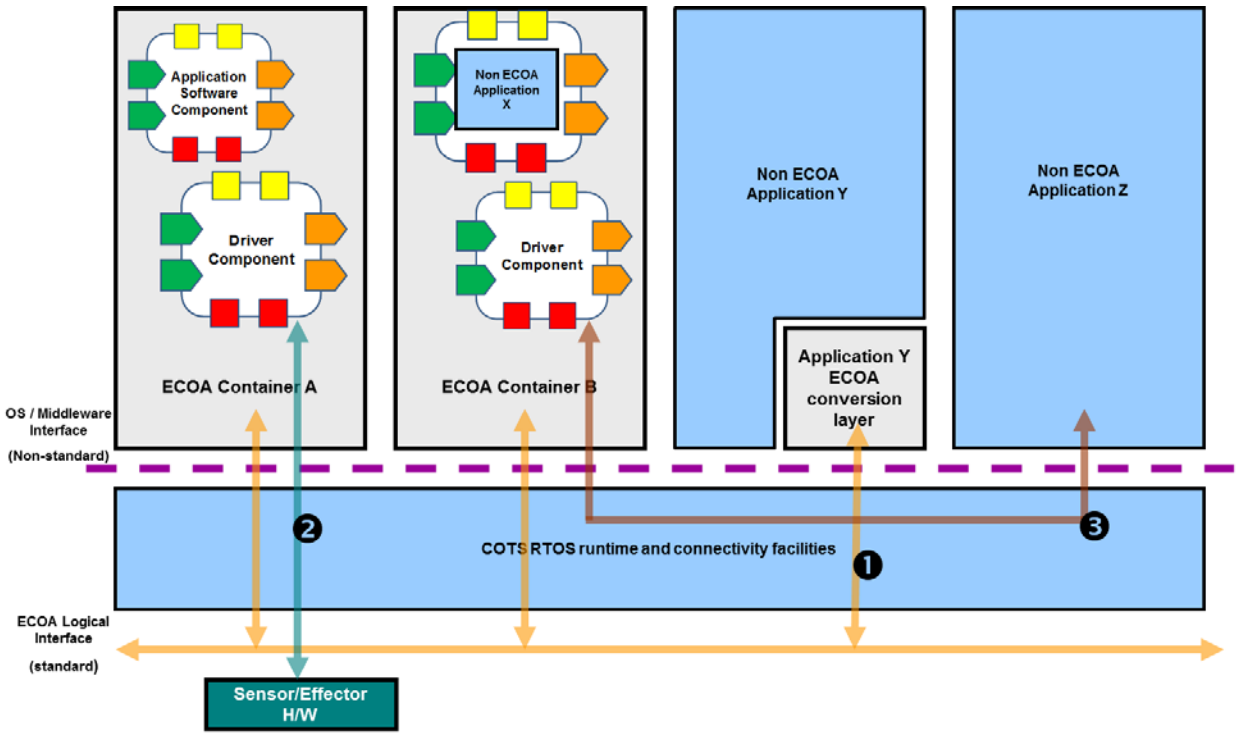

Figure 12 - Integration of Legacy Software and Hardware into an ECOA Architecture

9.1.1 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:

1. Wrapping or re-engineering as an ECOA Module that implements the necessary Inversion-of-Control behaviour (Application X in Figure 12)
2. 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 12)

3. 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 12) See the next section on this topic.

Method 1: Re-engineering to provide Inversion-of-Control, or “Componentisation”

> Legacy code will be placed within one or more functional Modules which act as ECOA compatible façades or wrappers. Together with a Supervision Module, these wrapper Modules will interact with the ECOA Infrastructure according to the IoC rules, while internally performing the mapping from ECOA-style execution to that expected by the legacy code. Ideally the design of wrapper Component, Module Operations and Services will be chosen to make the mapping straightforward.
>
> This option results in a fully fledged ECOA Application Software Component, lending it a limited degree of portability and allowing it to benefit from the same optimisations as other Components on the same ECOA Software Platform. For instance: avoiding the overhead of using communication channels and the ELI for interoperability within the same Protection Domain.

Method 2: Development of an ECOA Conversion Layer for a non-ECOA application.

> This is an option if the interfacing requirement is simple: i.e. small number of simple Services. It may be necessary in cases where the semantic gap between approaches adopted by ECOA and legacy code is large, or if the legacy application cannot be hosted in ECOA because its implementation depends on underlying technology (implementation language, RTOS etc) which is not supported by any ECOA Platform.
>
> The ECOA Conversion Layer must implement ECOA-conformant ELI message transmission (marked ❶ in Figure 12) and reception externally, translating these to legacy calls, data retrieval logic etc. internally. Connections marked ❷ and ❸ represent non-ELI message transmission and are discussed below.

9.1.2 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 12 : Container A communicates with a non-ECOA sensor/effector via the connection marked ❷ and for Container B to communicate 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 Containters (refer to 8.5), 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.

9.2 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 13.

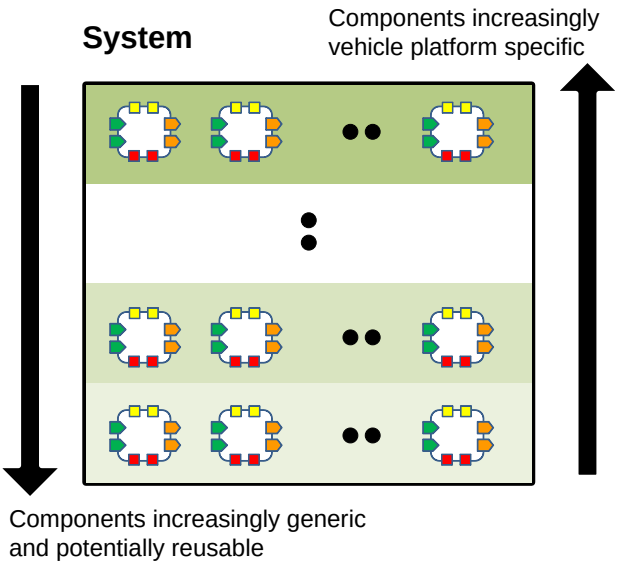

Figure 13 - Layered / Hierarchical Component Based Architecture

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, for example into Supervision Modules, should enable the remaining Modules to be more generic, and hence better candidates for reuse within other Components.

10 References

Ref.	Document Number	Version	Title
1.	IAWG-ECOA-TR-002	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume II Developers Guide
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 lV Common Terminology

Table 10-1 - Table of ECOA references