European Component Oriented Architecture (ECOA®) Collaboration Programme: Preliminary version of the ECOA Architecture Specification Part 1: Concepts

Dassault Ref No: DGT 2041078-A
Thales DMS Ref No: 69398915-035 --

Issue: 7

Prepared by
Dassault Aviation and Thales DMS

Note: *This specification is preliminary and is subject to further adjustments. Consequently, users are advised to exercise caution when relying on the information herein. No warranties are provided regarding the completeness or accuracy of the information in this preliminary version. The final version of the document will be released to reflect further improvements.*

Without prejudice to the property rights relating to the ECOA AS6 Standard, the information in this document relating to the changes envisaged for the transition from the ECOA AS6 Standard to the ECOA AS7 Standard is the intellectual property of Dassault Aviation and Thales DMS France SAS. The information set out in this document is provided solely on an 'as is' basis and co-developers of this specification make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.

This Page Intentionally Left Blank

This Page Intentionally Left Blank

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 multi-part Architecture Specification.

0.2 Main Introduction

The Architecture Specification provides the specification for creating ECOA®-based software systems. It describes the standardised programming interfaces and data-model that allow developers to produce ECOA® components and construct ECOA®-based software 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.

The ECOA Standard is modular. It is composed of a Core and Options, Extensions and programming language Bindings. This document deals with the Core, even if it may reference some Options or Extensions, in order to ease the understanding.

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of complex real time software systems using a component oriented 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 shall rely on the BNAE publications of the ECOA Standard.

3 Normative References

Architecture Specification Part 1	Dassault Ref No: DGT 2041078-A Thales DMS Ref No: 69398915-035 -- Issue 7 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	Dassault Ref No: DGT 2041081-A Thales DMS Ref No: 69398916-035 -- Issue 7 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	Dassault Ref No: DGT 2041082-A Thales DMS Ref No: 69398917-035 -- Issue 7 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	Dassault Ref No: DGT 2041083-A Thales DMS Ref No: 69398918-035 -- Issue 7 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	Dassault Ref No: DGT 2041084-A Thales DMS Ref No: 69398919-035 -- Issue 7 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	Dassault Ref No: DGT 2041491-A Thales DMS Ref No: 69398920-035 -- Issue 7 Architecture Specification Part 6 – Options
Architecture Specification Part 7	Dassault Ref No: DGT 2041086-A Thales DMS Ref No: 69398925-035 -- Issue 7 Architecture Specification Part 7 – Metamodel

4 Definitions

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

5 Abbreviations

API	Application Programming Interface
COTS	Commercial Off-The-Shelf
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
IoC	Inversion-of-Control
IT	Information Technologies
OS	Operating System
QoS	Quality of Service
RTOS	Real-Time Operating System
PINFO	Persistent Information
UML	Unified Modelling Language
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, so that the re-use of existing validation evidence is facilitated. The outcome should be a reduction of 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 (or unmodified) 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 open architecture approaches of modern IT (component-based, micro-service orientation, etc.) 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 component-oriented architecture using a model based approach, which provides the capability to formally specify applications, composed of an assembly of 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 (applications and computing platforms) and software assets (components), 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 interfaces. This promotes modularity, ease of integration, re-use and adaptability, each of which will impact significantly on the affordability of future military avionic systems.

Note: The concept of "service" is no longer present in the ECOA Architecture Specification. However, ECOA is yet fully compatible with service-oriented approaches and architectures, although it does not mandate them. Interactions modelled with ECOA can be implemented with various technologies, either service-oriented, event- or message-driven, data-based, etc. A specific Guideline document, [ECOA_SERVICE_ORIENTATION] details how a service-oriented architecture can be managed with ECOA.

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 within a 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 specific language bindings (for example a High Integrity Ada language binding). 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 of Applications or Components
- Scalability in terms of deployment of applications across multiple computing platforms and/or vehicles
- Commonality (of application software across platforms)
- Ease of deployment & integration of applications or 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 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;
- Compatibility and traceability with different communication protocols and architectural styles (service-oriented, event-driven, publish-subscribe, etc.);
- Integration of ECOA applications with non-ECOA applications.

6.4 Modularity of the ECOA Standard

6.4.1 Core, Options, Extensions and Bindings

The **ECOA Standard** is a modular one, which aggregates several sub-standards:

- The **ECOA Architecture Specification** (Core + Options), built on a set of fundamental features that constitute the **ECOA Core** and a set of optional features called **Options**. The **Options** are part of the metamodel but are not mandatory for a specific platform to be ECOA compliant.
- **ECOA Bindings**. The ECOA Component contract is translated in programming language APIs through programming language bindings, the **ECOA Bindings.** They are documented in specific documents that will reference the Architecture Specification and they are standardized on their own. To maximize the portability of a component implementation, a component provider should promote the use of a widely implemented binding.
- **ECOA Extensions**. Extensions are documented in specific documents that will reference the Architecture Specification.

The **ECOA Core** contains all the required features to be ECOA-compliant from a platform provider, a component provider or an application provider point of view.

An ECOA software artefact (component or application) may require **ECOA Options**; if this is the case, it will be usable only on Software Platforms that support these options.

For a given software artefact, the exclusive use of the **ECOA Core** guarantees maximum portability. The use of **Options** reduces portability. The use of **Extensions** further reduces portability.

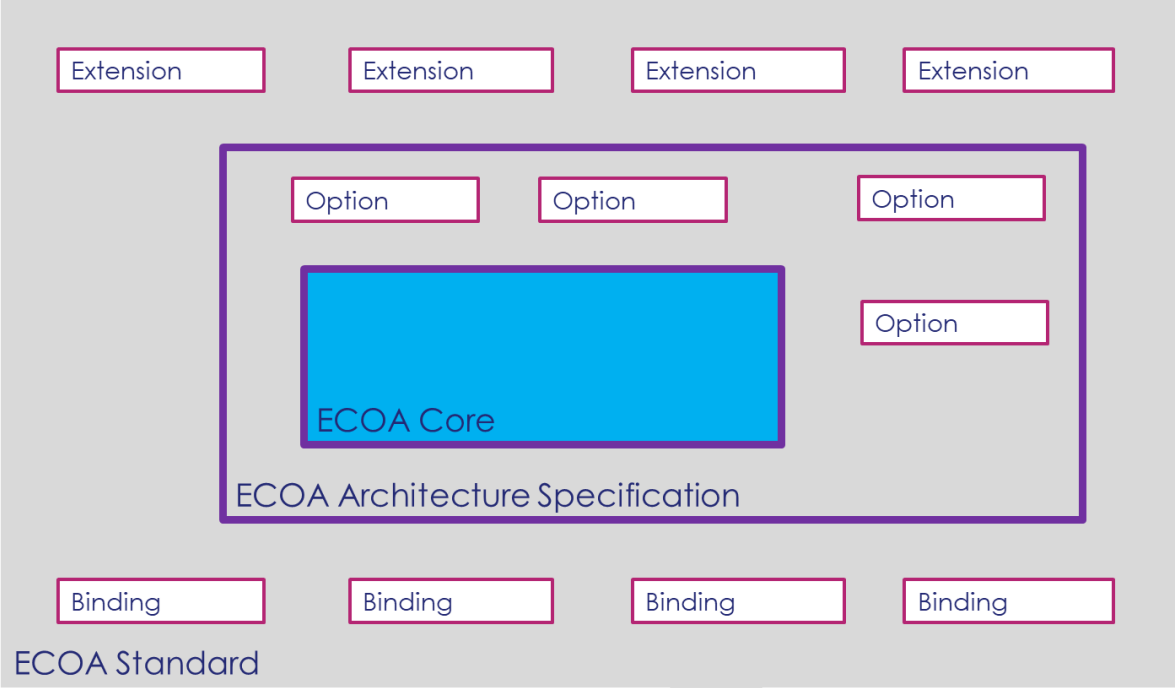

Figure 1 - ECOA Standard - Core, Options, Extensions and Bindings

6.4.2 Rationale for a modular standard

This modularity and extensibility of the ECOA Standard allows to accomodate different use cases and different user profiles:

a) **Use cases**
   1) Some use cases may require a simple and efficient software platform, possibly on a low-end hardware platform, for example because of size, weight, or power constraints.
   2) Other use cases may require a higher-performance hardware, allowing a more sophisticated software infrastructure to be used.

b) **User profiles**
   3) Some users may wish to acquire a "ready made", "off-the-shelf" platform, encompassing a maximum numbers of aspects. The use of a single model, covering all these aspects while remaining relatively simple, may be important to them.
   4) Other users may acquire or build a platform that is more tailored to a specific context (hardware, legacy software, etc.).

Therefore, through the modularity and the extensibility of the standard:

- The number of users that can be interested by the standard is maximised.
- Platform providers have the possibility to offer more or less feature-rich platforms.
- Platform providers have the possibility to offer more or less efficient and optimized platforms.

- Component and Application providers have the possibility to build a catalog of reusable components that may be used in many contexts, even if some components use extensions or options, and therefore will not be usable on all platform.
- The ECOA Architecture Specification is focused on software aspects only. It has no dependency on hardware. It is independent of system engineering aspects (methods, models, tools, etc.).
- New language bindings may be defined without impacting the ECOA Architecture Specification.

6.4.3 Architecture Specification documentation structure

The following figure shows how the different parts of the ECOA Architecture Specification are documented:

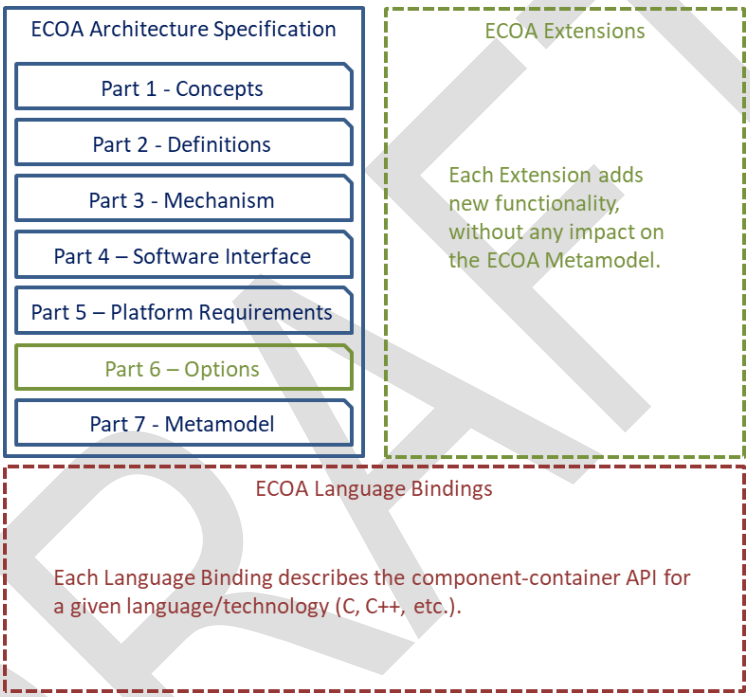

Figure 2 - ECOA Standard

Note 1: Architecture Specification Part 5 is the document that contains the full list of options that make up the ECOA Architecture Specification.

Note 2: Architecture Specification Part 6 is dedicated to the documentation of the ECOA Options.

6.5 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

- Applications or Components, in a rigorous, machine readable manner that includes what is provided, what is required in support, and optionally aspects of "quality of service". Using this, the suitability of a component or an application 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 executables in an integrator-defined configuration, resulting in ready-to-run applications.
- Execution of applications 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 formal manner (e.g. using XML).
- Optionally, through extension, 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 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, persistent information access and optionally error handling.

Concerning Safety & Security

- The definition of a High Integrity Ada language binding
- The capability to define a new Rust language binding
- The ability to capture temporal properties or Quality of Service attributes through optional meta-data

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 3 is a diagram summarising the main artefacts of ECOA.

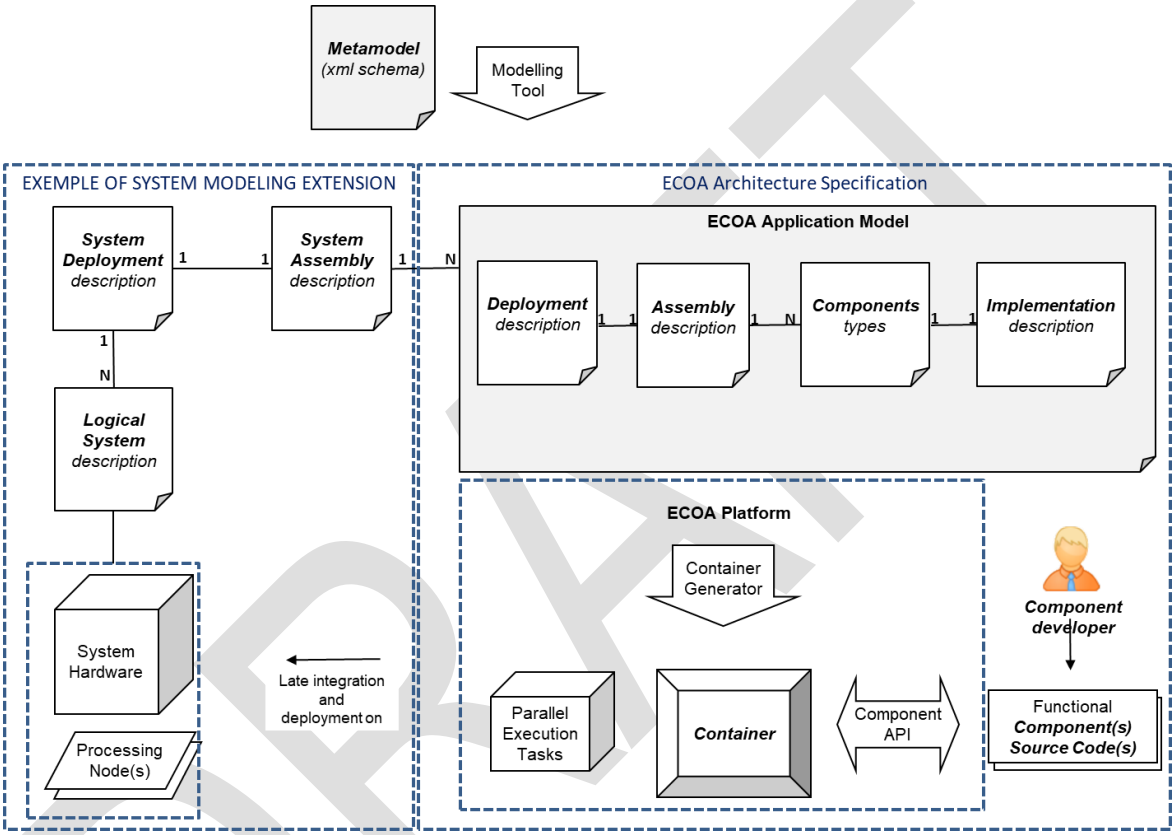

Figure 3 Introductory Overview of ECOA Artefacts

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 Architecture Specification. 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.

In ECOA models, a software ***Application*** is described and organised into a set of collaborating ***Components*** called an ***Assembly***. The Components collaborate by invoking **Operations**. Component Operations may be grouped and connected as groups, using **metadata** and high-level linking capacities. This allows by example to implement the notion of micro-services within the Application.

The ECOA model also describes, in a ***Deployment*** part:

- How the ***Components***, composing an ***Application***, are deployed on ***Tasks***, expressing the potential parallelism of execution of those ***Components*** instances
- How an ***Application*** will communicate with the rest of the world though communication ***Ports***

It is intended (but not mandatory) 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(s) is integrated with the ECOA Components (functional components that the application developer creates) late in the development lifecycle. This means that if either Containers or Components change, such as during redeployment or after functional change, there is less expenditure of effort re-establishing a complete software system.

7.1 Applications and Components

From the top-down perspective of overall system architecture, software design in ECOA is expressed in terms of ***Applications*** and ***Components*** composing it. Two major points have to be known to well understand the ECOA Standard philosophy:

- An ***Application*** is implemented through an ***Assembly*** of ***Components***, and a ***Deployment*** that specifies the technical information necessary to build an executable software.
- A ***Component*** may be implemented through an ***Assembly*** of ***Component*** instances (***Composite***), or directly through a specific programming language. In other words, the Component concept is recursive, or hierarchical.

A basic definition of a ***Component*** is that it is a building block of a software system. The power of the polymorph ECOA ***Component*** is the ability to play different roles it fulfils through its different forms:

- From an overall System architecture perspective, the ***Application*** form of the ECOA ***Component*** is used to build a Multi-Applications system and is the unit of exchange for "pre-integrated" or "ready-to-run" software artefacts. Optionally, the assembly between applications can be formalized with models through an extension.
- From an ***Application*** software Architect / Integrator perspective, an ECOA ***Component*** (of a lead Component or of a Composite) will fulfil the key role of units of exchange between software developers and/or integrators. Compared to an Application, a Component is a software artefact that is not yet integrated with others. For example, the real-time architecture is not yet defined. The integration work remaining to be done is higher than for an Application.

As an Application is a Component, it has an interface defined in terms of ECOA Operations. These operations are mapped to abstract communication Ports, which are in turn, mapped to a communication protocol. Thus, communication between several Applications is possible, and is entirely managed by the ECOA Platform if its supports the chosen protocol.

An Application is generally exclusively composed of Components. But, if the Platform supports it, il may also be a hybrid application, made of ECOA Components and non-ECOA or legacy software.

Table 1 Summary of differences between Application and Component

	Application	Component
Made of:	Assembly of Components + Deployment	• Itself (source or binary code) • or Assembly of Components
Delivered as:	Executables	Source or binary code
Ready to run?	Yes Software work is done. Some configuration of communications may be necessary/possible.	No Real-time architecture not yet defined. Some software work remains to be done.
Communicates through	ECOA Operations, mapped on Ports, which are mapped to network protocols (defined though Extensions or Platform specific features)	ECOA Operations, managed and optimized by the ECOA Platform (e.g. shared memory when possible)
Communicates with	other Applications or anything that respects the given protocols	other Components within the Application or outside the Application

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

An ECOA ***Component*** defines its interface through a set of input/output ***Operations***. A ***Component*** considered from an external point of view, independently of any particular implementation, is called a ***Component Type***.

An ***Operation*** is either, a ***Data Operation***, an ***Event Operation*** or a ***RequestResponse Operation***. Each one of these types of operations correspond to an interaction mechanism. Components collaborate through logical links between their Operations.

An ECOA ***Component*** may be tailored through ***Components Properties***.

The following figures represent a Component graphically, with its Operations and Properties.

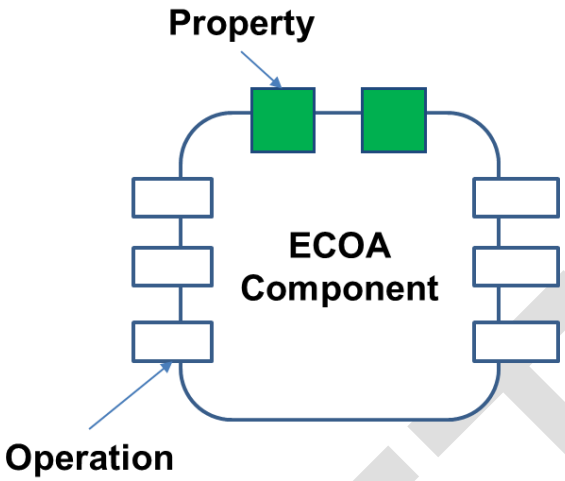

Figure 4 Abstract View of an ECOA Component - Operations not detailed

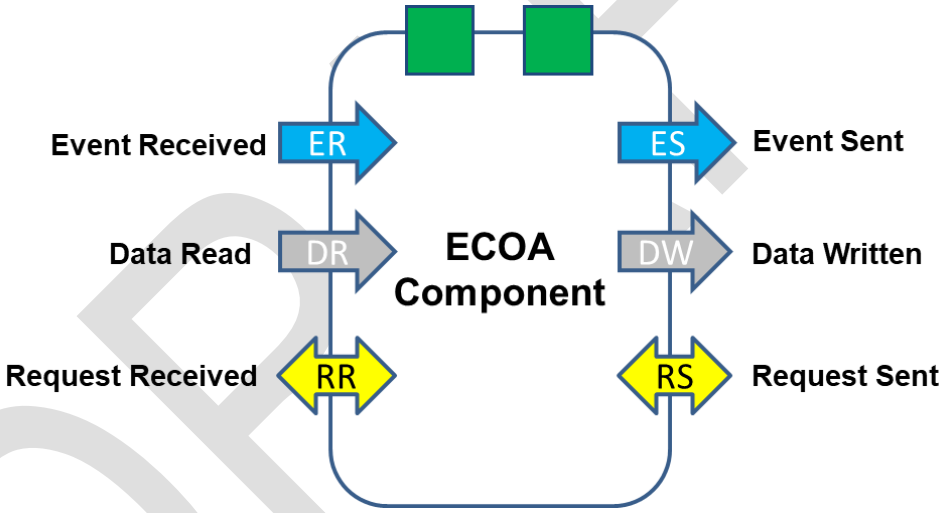

Figure 5 Abstract View of an ECOA Component, showing the different types of Operations

An ECOA ***Component Type*** may also include other elements (like PINFOs, Triggers, etc.), that are not represented here for the sake of simplicity.

ECOA offers means to mimic service-oriented approaches by functionally linking together operations of a component (see ImplicitLinks and metadata in Architecture Specification Part 7).

Architecture Specification Part 3 *contains a full description of the Operations that can be used to make up a Component 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 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 ***Component*** Operations in response to the actions of other ***Components*** or external stimuli. 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 function from implementation;
- To focus a software implementation on the function 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.

Note : 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 of the ***Components*** that it hosts. The Container and ***Component*** 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 a ***Component*** and the ***ECOA Software Platform***.

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

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

ECOA mandates Components 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 **External Components**, which are described in section 7.6.2, are allowed to bypass these design principles.

7.3 Component Implementation and the Container Interfaces

A declarative entity in the ECOA XML Metamodel called a ***Component Implementation*** provides a way to specifies different implementations that may exist for the same ***Component Type***: perhaps targeting different hardwares; perhaps written in different languages…

Component Implementation developers can conceivably use any technology, languages, libraries etc. they need to get the job done. However, to guarantee portability and reusability a ***Component Implementation*** 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 ***Component Implementation*** can be developed. Cross-compiled object code may be delivered to software system

integrators who can compose ***Component Implementations*** from different suppliers together to form their ***Application***, without the need to share source code or header files.

Following the Inversion-of-Control principle, ECOA ***Component Implementation*** are passive and (with some specific exceptions detailed in 7.5) do not assume control over, or even knowledge of, the wider software, but instead are hosted and operated under control of the ***ECOA Software Platform***.

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

The ***Component Implementation*** is derived from the ***Component Type***. ***Component Implementation*** software entry points are handlers for ***Operation*** invocations coming via the Container from the wider ECOA ***Application*** (or external port).

Methods exposed by the Container Interface provide the means by which ***Component Implementation*** can call ***Component Operations*** on other ***Components instances***, or make use of Infrastructure services. A Container Interface may be mechanistically derived (code generated) in its entirety from the Component Type and Component Implementation and supporting declarations.

Infrastructure facilities provided by the Container Interface include time services, logging and fault management.

Where a ***Component Implementation*** requires an internal state to be maintained, it shall use a ***User Context*** and an optionally ***Warm Start Context*** ([OPTION WARM START CONTEXT]). When using these contexts, the component supplier can expect the platform to maintain them.

Note : Architecture Specification Part 4 defines the Component User Context and the Component Warm Start Context.

7.4 Component Lifecycle

The Infrastructure initialises starts and stops the ***Components*** Instances implementing an ***Application***. The infrastructure achieves this by sending **Lifecycle Events** to each Component Instance.

When a ***Component*** Instance is not in the RUNNING state, any activation requests which are not lifecycle events are ignored (received events, received requests, data notifications….).

The Infrastructure will generally control the lifecycle of ***Component*** Instances in response to higher level system events that affect the lifecycle of the ***Application*** as a whole (e.g. faults, external interfaces…), or through Supervision Components ([OPTION SUPERVISION]).

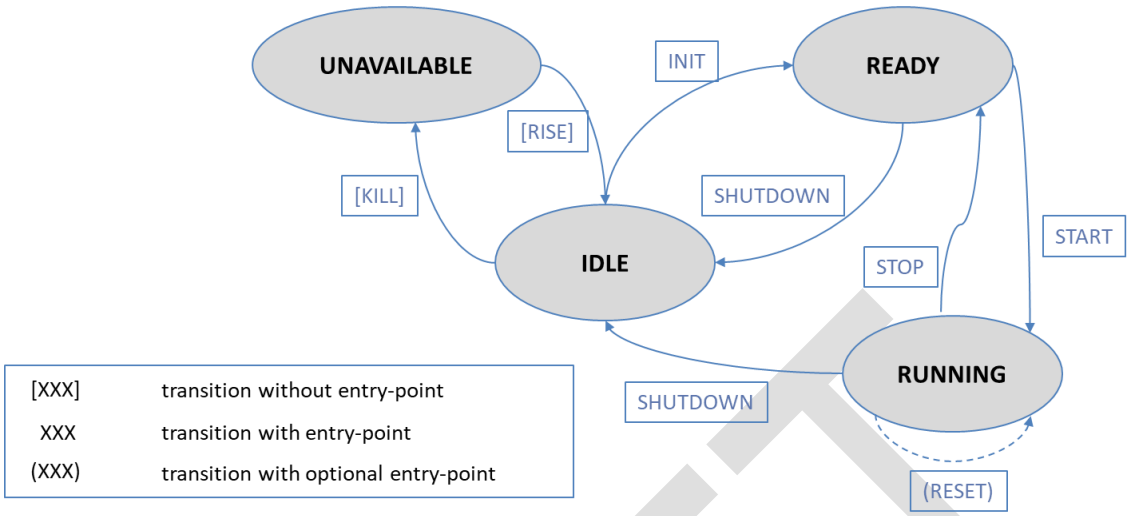

Figure 6 - ECOA Component Lifecycle

7.5 Persistent Information

ECOA provides a way for ***Components*** 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.

ECOA provides an interface for Components to access PINFO through, read and seek operations. It is not possible for Components to access PINFO in write mode excepted if the ECOA platform supports the [OPTION PINFO WRITE].

Uses of PINFO could be:

- For an Component to read functional initialization data, such as sensor performance look up tables,
- For several Components to read initialization data, such as public mission planning data.
- For a Component to save a configuration or any non-volatile information (if the platform supports the [OPTION PINFO WRITE]),

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

NOTE: The use of PINFO WRITE and PINFO READ in the same application may be restricted in order to avoid unwanted hidden communication channels.

7.6 ECOA Special Component Types

7.6.1 PeriodicTriggerManager

Given the general IoC principal that ***Components*** are entirely passive, the question arises as to how the developer should create active ***Applications***. A special Component Type called a ***PeriodicTriggerManager*** provides a mechanism to allow this.

A ***PeriodicTriggerManager*** is defined through a Component Type, then instanciated as a classical ***Component Instance*** and wired to other ***Components Instances*** like any ***Component Instance***, but its

implementation is provided by the Infrastructure. Using a ***PeriodicTriggerManager*** Instance an Application 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 Component.

7.6.2 External Components

An ***EXTERNAL component*** type is a specific type of component that will manage external interfaces that are not defined in ECOA models (connection to a specific data bus, management of a specific HW, etc.). This kind of components usually need to have a dedicated task to wait, for example, for messages coming from the non-ECOA world. This kind of components is the only one that will not respect the mono-threading rule and that will have a dedicated additional thread, called the “external” thread. This additional thread is automatically created by the infrastructure. The communication and synchronisation between the "external" thread and the standard thread, if required, may be done using several solutions; for example: ECOA operations, atomic variables, lock-free data structures, non-ECOA mechanisms provided by the OS (such as mutexes).

The ***EXTERNAL components*** must behave as standard ECOA Components in all interactions with their respective Containers, but they may, internally, use legacy (e.g. OS and hardware) interfaces to communicate with legacy devices. Such components will therefore be less portable than pure ECOA ***Components***.

7.6.3 Supervision Components

This type of components is specific to [OPTION SUPERVISION].

A Supervision Component is a specific type of component that allows to control the lifecycle of ***Component Instances*** within an ***Application***.

It also enables and disables conditional links between components.

7.6.4 DynamicTriggerManager

This type of components is specific to [OPTION DYNAMIC TRIGGER].

A DynamicTriggerManagerComponent is a specific type of component that allows to receive an event, and send it back at specific date in the future.

While this behaviour can be implemented as a standard component, there is no code to develop for a DynamicTriggerManager: its implementation is entirely provided by the Infrastructure.

7.7 Hardware and Software Interoperability

The preceding sections describe the declarative entities and concepts required to define a component-oriented software 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 ***Component Implementation*** 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.

7.7.1 Interoperability Protocol

Communications between ECOA ***Applications*** may be achieved through the use of a well defined interoperability protocol.

Use of an interoperability protocol through a network ensures that independently developed ECOA Software Systems or ECOA Stacks are able to work together.

Interoperability Protocol may be defined through an ECOA Extension.

7.7.2 Driver Components and Legacy subsystems

It is one of the key objectives to be able to deploy ECOA Applications on non-ECOA legacy platforms and to be able to integrate non-ECOA software and hardware with an ECOA System. Figure 7 shows different cases involving integration of legacy subsystems which are discussed further, below.

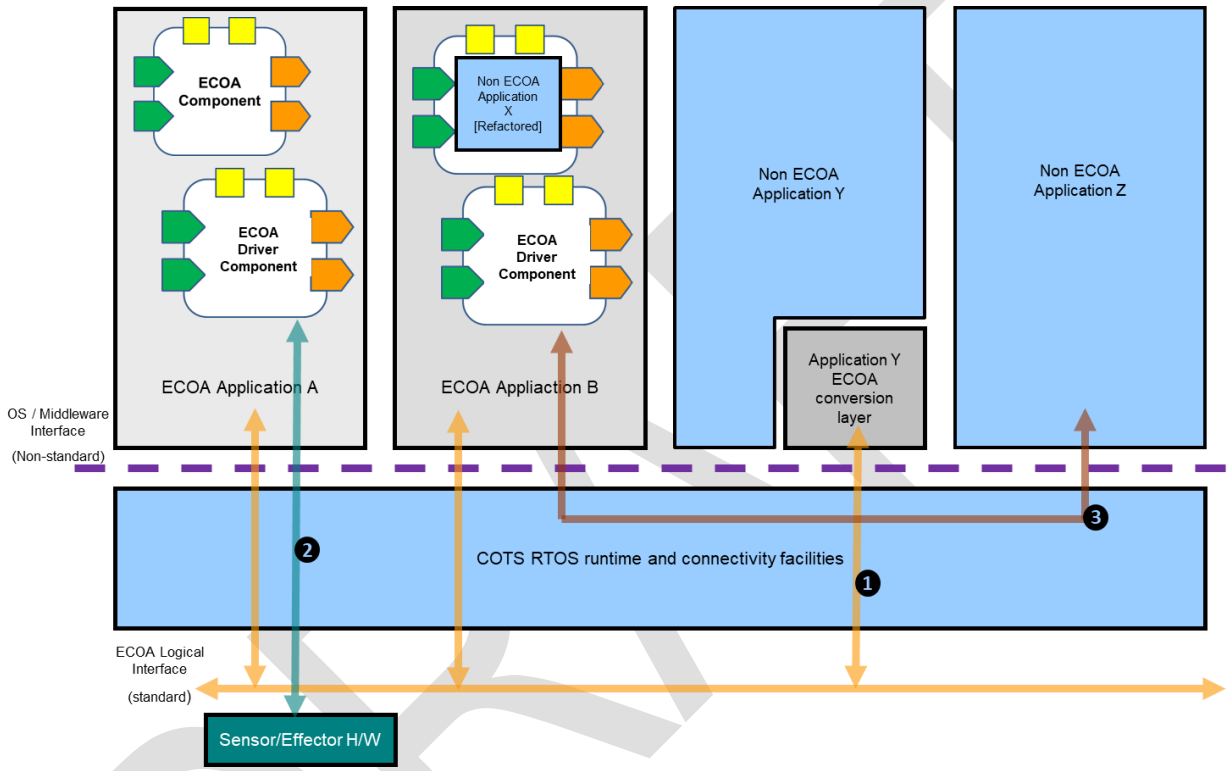

Figure 7 Integration of Legacy Software and Hardware into an ECOA Architecture

7.7.2.1 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 ***Component***. This may imply the inability to effectively decompose application software into cohesive ***Components***, 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 ***Component*** that implements the necessary Inversion-of-Control behaviour (Application X in Figure 7)

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 7)

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 7). See the next section on this topic.

7.7.2.2 Driver Components

The notion of a Driver Component is introduced to describe a Component that translates the interface protocol used by legacy hardware or software into operations specified in a Component Type with an ECOA description and behaviour.

Driver Components must behave as standard ECOA Components in all interactions with their respective Containers, but they may, internally, use legacy (e.g. OS and hardware) interfaces to communicate with legacy devices.

Such Driver Components will therefore be not portable across ECOA Software Platforms, or less portable than pure ECOA Components.

Usually, External Components (§7.6.2) are also Driver Components, because they use non-ECOA interfaces.

The notion of Driver Component is not formalized in the metamodel, and is not relevant for the Platform. It is only a way to describe "hybrid" Components that rely on other interfacing mechanisms that those defined by ECOA. There is no platform support for Driver Components.

7.8 ECOA MetaModel

7.8.1 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 software system designer may gain confidence in a design, and to do so well in advance of final integration: i.e. that the software system, once completed, will meet its functional and non-functional requirements.

Optional QoS attributes may be used by tools to check compatibility between needs and available solutions.

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** and **Applications** 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.

7.8.2 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 the table below[1].

Table 2 XML files used for system description

Data Type Definitions:	Describes the data types used in ***Operations***
Component Types:	Describes a Component from an external point of view. Identifies ***Operations***, Properties and PINFOs.
Component Implementations:	Defines a specific ***Implementation*** of a ***Component Type***
Assembly:	Defines all the ***Components*** instances and links between those ***Instances*** composing an ECOA ***Application***
Deployment:	Maps ***Components Instances*** onto ***Task***, expressing the potential parallel execution of ***Components*** in an ECOA ***Application***. Maps external ***Operations*** onto ***Ports***.

As can be seen, much of the ECOA XML Metamodel is given over two aspects of ***Component Definition***, and ***Component 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.

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