European Component Oriented Architecture (ECOA®) Collaboration Programme: Guidance Document: Developer's Guide

Date: 27/11/2017

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

1 Table of Contents

2 List of Figures

3 List of Tables

No table of figures entries found.

4 Abbreviations

API	Application Programming Interface
ARINC	Aeronautical Radio INC
ASAAC	Allied Standard Avionics Architecture Council
AS	Architecture Specification
ASC	Application Software Component
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
ELI	ECOA® Logical Interface
GFX	Government Furnished Equipment
OS	Operating System
POSIX	Portable Operating System Interface
QoS	Quality of Service
UDP	Unreliable Datagram Protocol
WCET	Worst Case Execution Time
XML	Extensible Mark-up Language

5 Executive Summary

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

This document is a guide to developers of ECOA® Applications and ECOA® Platforms.

As a precursor to this document, the reader is encouraged to familiarize themselves with the basics of ECOA, set out in the following parts of the ECOA® Architecture Specification (AS): Concepts Document [AS-Part 1], Definitions [AS-Part 2] and Mechanisms [AS-Part 3].

This document should be read in conjunction with the with the ECOA® Software Interface [AS-Part 4], together with ECOA® Language Bindings for C [AS-Part 8], C++ [AS-Part 9], Ada [AS-Part 10] and High Integrity Ada [AS-Part 11]

The main parts of this document are:

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

6 Introduction

The Architecture Specification (AS) provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA-based system. It is introduced in [AS-Part 1] Concepts and uses terms defined in [AS-Part 2] Definitions. For this reason, the reader should read these documents, prior to this document. A list of the parts comprising the Architecture Specification can be found in Section 9.

This document identifies the activities and considerations to be applied when developing software using ECOA concepts. The document is intended to be understood by developers who have little experience of using the ECOA paradigm.

Specific guidance is available on the ECOA website [WEB-1] on a number of topics, including:

- Time Synchronisation,
- System Management (including Fault Management and Lifecycle),
- Reconfiguration (context management, warm/cold restart),
- Transports Binding (ELI connection),
- Container Level Checking (QoS, data range),
- Data Servers,
- File Access using File Servers and Driver Components,
- Insertion Policies,
- Module Behaviour.

In addition, there is tutorial material providing worked examples of ECOA systems, available from the ECOA website [WEB-1].

This document is divided into two main sections:

- Section 7 provides a description of the development activities, identifying what considerations the ECOA paradigm requires a developer to make. This is described under two subsections:
    - Section 7.1, which provide guidance for developing application software,
    - Section 7.2, which provides guidance for developing an ECOA software platform.
- Section 8 identifies the rules that must be followed both during development and when producing metadata (XML) which is used when exchanging aspects of your system with third parties.

The intended audience for this document is:

- System Designer, refer to section 7.1.2 & 7.1.4,
- Software Developer implementing an ECOA Application Software Component, refer to Section 7.1.3,
- Software Developer implementing an ECOA Software Platform, refer to Section 7.2,
- Software/System Integrator, refer to section 7.1.2 & 7.1.4,
- Software/System Tester, refer to Section 7.1,
- Software Team Managers, refer to Section 7,
- Software Process Developer, refer to Section 7.1.

7 Software Development Activities

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

This section has two subsections:

- The first provides top level guidance on the development of ECOA Systems, describing the activities entailed in designing and developing ECOA Application Software Components (ASCs).
- The second provides guidance on the development of ECOA platforms.

7.1 Software Development Guidance

7.1.1 Overview

This section describes the activities to be undertaken during the following development phases:

- Software System Design,
- Component Development,
- System Integration.

Figure 1, illustrates the relationships between the development activities.

In the context of this guidance, 'system' is a software system, which executes across one or more ECOA Platforms.

ECOA does not attempt to define or impose any functional architecture on the software system. Developers can therefore develop ASCs bespoke to the system under development without regard to similar systems being developed elsewhere. Although ECOA does not define a functional architecture it does provide a framework which supports reusable components based on such a Reference Architecture.

It is anticipated that domain Reference Architectures will be agreed in the near future; these are expected to specify the functional behaviour for systems and ASCs, and to establish data models. It would be advantageous for the System Designer to be aware of these architectures, as they should promote an increase in the availability of COTS ASCs.

In the absence of an applicable domain Reference Architecture, the System Designer should provide a project Reference Architecture to provide guidance for the ASC developers.

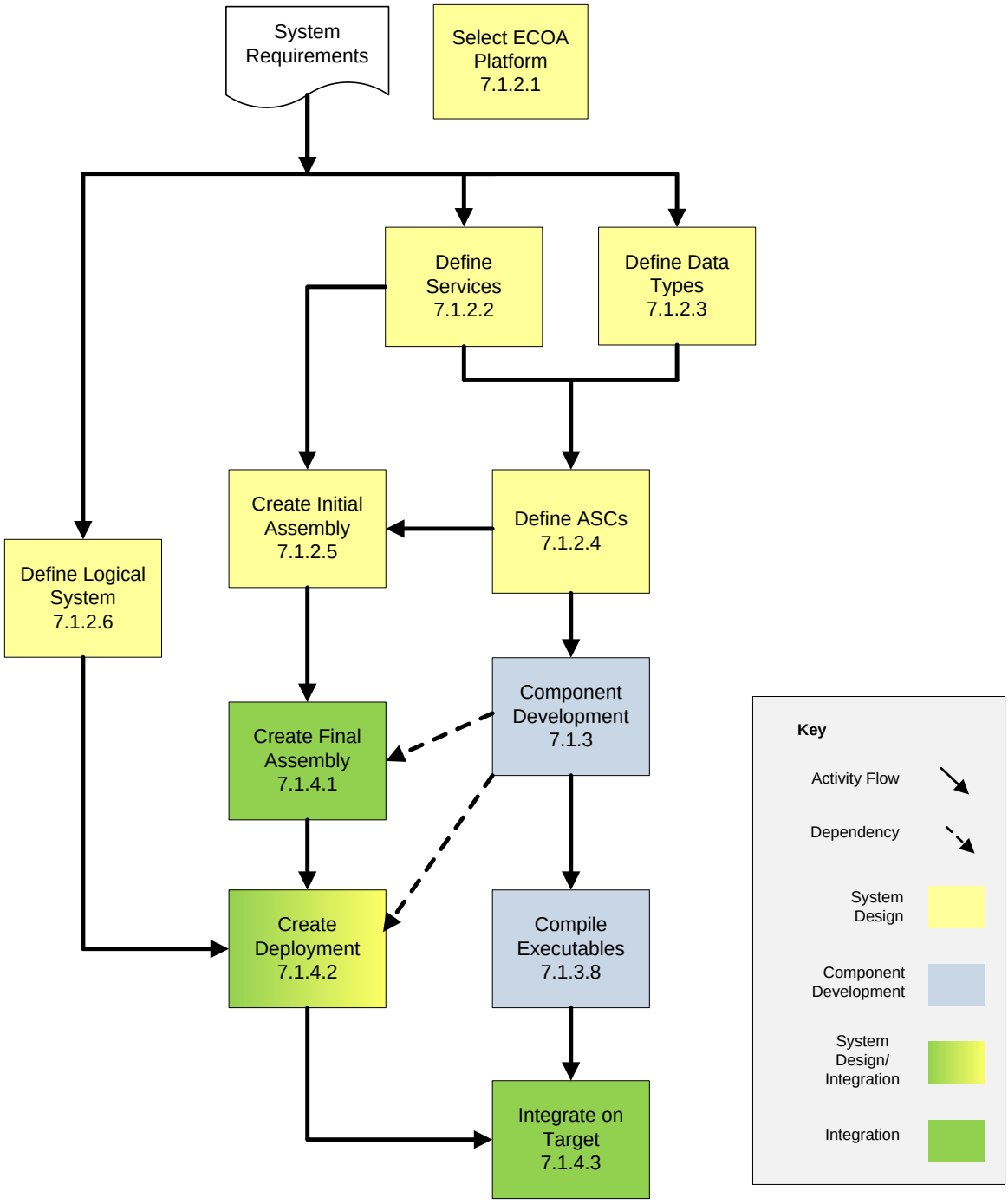

Figure 1 – Software Development Activities

7.1.2 Software System Design

Standard software systems are comprised of one or more Applications which provide the required functionality. These applications may be partitioned in Protection Domains and may execute across a number of computing nodes.

The inputs to the design of an ECOA software system are the usual inputs defined for any software intensive system:

- Functional Requirements,
- Non-functional complementary requirements.

When employing the ECOA paradigm, software systems are constructed using one or more ECOA Application Software Components (ASCs), whose interactions are described through Services.

An ASC or group of ASCs (composite) could be regarded as an Application and may be deployed in a similar manner.

The System Design phase has the following activities:

- Select ECOA Platform,
- Define Services,
- Define ASCs,
- Define Data Types,
- Create Initial Assembly,
- Define Logical System.

Some of these activities occur in parallel, and all of the activities are likely to be performed iteratively throughout the design.

It is assumed for the purposes of this guidance that the functional, interface and complementary (non-functional) requirements for the software system have already been established.

While this guidance relates to the software system development, it is anticipated that the Software System Designer will have a role in the selection of computing platform(s) on which the system will execute, see Section 7.1.2.1.

The initial activities of defining the ASCs in terms of Services and Data types, and creating the Initial Assembly are independent of the hardware architecture.

7.1.2.1 Select ECOA Platform

An ECOA platform is the hardware and software infrastructure on which the ASC implementation is hosted.

Whilst it is possible that the Software System Developer could decide to develop their own ECOA Platform (refer to section 7.2 for guidance), the ECOA Librarian should be able to provide contact details for providers of suitable ECOA Platforms which have already been developed.

When selecting an ECOA Platform, the Software System Designer will need to consider a wide range of issues including:

- Interfaces to be supported, including hardware and software,
- Performance requirements,
- Certification requirements,
- Legacy software to be integrated, including software libraries as well as other already developed non-ECOA software,
- Software language support by the provided Code Generator,
- Scheduling policies supported.
- Support for optional ECOA platform features (e.g. ELI, fault handler error and recovery actions)

These considerations will influence the choice of processing hardware and architecture that will need to be supported by the ECOA Platform provider.

In addition, Safety and Security certification will need to be considered. ASCs requiring higher level of certification will need to be partitioned, either via hardware or the OS from lower certification level ASCs.

The certification level to which the ECOA Platform software infrastructure has been developed will need to be considered. In addition, any ECOA Platform tools including the Code Generator supplied by the platform provider will need to be developed to the appropriate Tool Qualification Level.

The developer's preferred software implementation language (e.g. Ada, C or C++), will need to be supported by the supplied Code Generator.

7.1.2.2 Define Services

This activity is performed iteratively along with "Define Data Types", "Define ASCs" and "Define Initial Assembly".

ASCs are interfaced via Services with other ASCs. A Service in ECOA comprises a cohesive group of Service Operations through which the ASC (client) may, via its Required Service, access the Provided Service of another ASC (server). Service Operations may take the form of publish/subscribe Versioned Data access or Event or Request-Response exchanges between connected ASCs [AS-Part 1].

It is not essential, but may be beneficial, especially for larger systems, to perform a Use Case analysis of the system requirements, the resulting Use Cases can provide context for the functional requirements.

Using the requirements and Use Cases, the designer identifies the Services to be provided by, and the Services required of the System. Successive/iterative decomposition of Use Cases leads to the further identification of Services within the System until their provision can be delegated to an ASC.

It is recommended that a Service should be as autonomous as possible; it should not require a client ASC to use other Services to fulfil a specific function. In addition, Services should be specified for stateless interactions, i.e. a Service request should contain all the information necessary (from the client ASC) to complete that Operation, and be independent of the invocation sequence of other Operations.

The granularity of a Service should be considered, a Service could be considered too coarse-grained where:

- It contains too many Operations,
- It exchanges more data than needed.

Coarse grained Services present a more complex interface to a client ASC, and are prone to more change.

Alternatively, a Service could be considered too fine grained where:

- It contain insufficient Operations
- It exchanges insufficient data.

Too fine grained services require the client ASC to make multiple Service calls with an inherent performance impact.

A balance is therefore required between level of abstraction, likelihood of change, complexity of the Service, and the desired level of cohesion and coupling. A trade-off needs to be made while taking into account non-functional requirements particularly performance.

The Operations of a Service should be:

- Coherent, i.e. addressing a related purpose,
- Sufficient, i.e. avoiding the need to use two (or more) Services to invoke all the necessary Operations for a specific function.

ECOA supports the following operation types (see also section 7.1.3.3.1 for more details):

- One-way (Event):
  this sends a command, with or without data, to one or more recipients. It should be considered where no response is required from the receiver(s).
- Two-way (RequestResponse):
  this operation type should be considered where a response is required from the server. The response may be synchronous or asynchronous.
- Independent push/pull (VersionedData) (see also Section 7.1.3.3.3 for more details):
  this operation type should be considered where the data is to be published to multiple recipients.

Transactional analysis should be used to determine the most suitable operation type.

Refer to [AS-Part 7] for a description of the Service attributes. Outside of a Software Development Tool the service definitions are captured in the “interface” XML file [AS-Part 7].

Additional consideration should be made to include any “non-functional service operations”, such as those which can provide information about the availability of the service. The inclusion of such operations is a system design choice and may not be required.

7.1.2.3 Define Data Types

This activity is performed iteratively with “Define Services”, “Define ASCs” and “Define Initial Assembly”.

It is recommended that a Reference Architecture’s data model appropriate to the system domain is referred to when specifying the data types to be used.

ECOA provides a range of basic data types [AS-Part 4], which can be used to create more complex types. These types can then be used when defining the data to be communicated via the Services.

7.1.2.4 Define ASCs

This activity is performed iteratively with “Define Services”, “Define Data Types” and “Define Initial Assembly”. This activity is undertaken independent of any hardware architectural considerations.

It is recommended that a Reference Architecture appropriate to the system domain is used when mapping the identified Services to ASCs.

The designer should initially consider the reuse of any ASCs developed for previous ECOA projects, which may satisfy the service requirements. In addition, the ECOA Librarian would retain lists of ASCs which may be utilised, these may be GFX or subject to purchase/licensing agreement.

Where there are no existing ASCs providing the required Services, or where the provided Service Operations are not compatible with respect to data types or Quality of Service (QoS), consideration should then be given to the modification or extension of existing ASCs. In the event of reuse not being possible, then a new ASC will need to be developed.

If an ASC is to be developed or modified by a different company to one performing the software system design, then the appropriate specifications must be produced for the ASC Supplier. These specifications include: identification of the services to be provided, the QoS requirements for those services, control requirements for the component (lifecycle), constraints on faults.

For ASCs being developed in-house the same level of specification is still required, but if an integrated system/software development toolset is being employed then there should be no need to export the Interface specification from the toolset.

Refer to [AS-Part 7] for a description of the ASC specification attributes. Outside of a Software Development Tool, the ASCs are specified in the “componentType” XML file [AS-Part 7].

How the ASCs are managed both functionally and with respect to Health Management may be addressed by the Reference Architecture. Refer to section 7.2.11 for a description of Fault Management provided by an ECOA Platform. ECOA is capable of supporting a wide range of different approaches with respect to system management; refer to System Management Guidance [WEB-1] for more detail.

If integration with non-ECOA domains is required this could require additional ASC(s) to be defined which convert non-ECOA communications to services, see section 7.1.2.7.

When designing large systems, it may be beneficial to aggregate ASCs into Composites [AS-Part 3] to reduce complexity when considering the system design.

A Composite is a functional grouping of ASCs, and is similar to an ASC in that it has provided and/or required Services, which are ‘promoted’ from the Services of its internal ASCs. Composites and the promotion links are a concept to reduce complexity through hierarchical assemblies.

ASCs and Composites are both logical design concepts which have no existence at technical levels, refer to section 7.1.4.1.

7.1.2.5 Create Initial Assembly

The Initial Assembly describes the structure of an ECOA software system, independently of its deployment onto a Logical System (see section 7.1.4.2). The Initial Assembly consists of:

- ASC instances,
- Service Links (also known as Wires),
- Optionally Composites.

One or more instances of ASCs are created from the Component Definitions; each has a set of instantiation parameters known as Properties and a set of instantiation PINFOs. The behaviour/functionality of each instance of an ASC can be tailored through its Property values.

Wires are used to connect ASC Services between instances of ASCs within the same assembly. Each Wire connects a Provided Service to a Required Service.

7.1.2.6 Define Logical System

This activity defines the Logical System [AS-Part 7], which is a logical view of hardware elements which constitute the Software System: these are Computing Platforms, Computing Nodes and Processor, together with how they are linked.

If the ECOA System consists of a single ECOA platform, the Logical System may be taken directly from the Platform Supplier specification.

Where the Software Development Tooling has the appropriate facilities, early verification of the design should be possible using the Logical System and Initial Assembly. This would verify that the high level system functional requirements are being satisfied, and given an initial consideration of the ASC deployment, provide an assessment of the processing power required of the platform hardware. This analysis can also be used to allocate processing time to an ASC, which will be captured in the requirement specification for the ASC to be satisfied by the developer.

Outside of a Software Development Tool, the Logical System is defined by the “logical-system” XML file [AS-Part 7].

7.1.2.7 Integration with Non-ECOA Domains

The following paragraphs outline methods the System Designer can consider for integration with non-ECOA domains:

- **Re-engineering to provide Inversion-of-Control, or “Componentisation”.**
  The non-ECOA code is re-written to comply with an ECOA tasking model, and is integrated into the ECOA Module(s) implementation of an ASC.

  If the non-ECOA code being integrated executes tasking Operations or has non-ECOA dependencies, this will result in the ASC being classified as a Driver Component, see section 7.1.3.3.4, for further details.

- **Development of an ECOA Conversion Layer for a non-ECOA domain.**
  This is an option where:
    - the interfacing requirement is straightforward, e.g. small number of simple Services,
    - the semantic gap between approaches adopted by ECOA and legacy code is large,
    - the legacy application cannot be hosted in ECOA because its implementation depends on underlying technology (implementation language, RTOS etc.) which is not supported by the ECOA Platform.

  In this case an ECOA Conversion Layer on the legacy software system must implement ECOA-conformant ELI message communications [AS-Part 6].

- **Develop a Driver Component**
  This component would interface directly (outside of the ECOA framework) with the non-ECOA Domain. It would be responsible for translating the non-ECOA communications to Service Operations for use by the rest of the ECOA system. In this case the communication with the non-ECOA domain need not be conformant with the ELI [AS-Part 6]. This option will likely be preferable where the target / procured ECOA platform has no support for ELI.

7.1.3 Component Development

7.1.3.1 Overview

The guidance in this section is applicable to both an ECOA ASC Supplier and to an in-house Software Developer with respect to the implementation of an ASC.

Traditionally software development involves writing application code that conforms to the System Design. The ECOA paradigm is no different in this respect except it specifies a set of rules relating to the implementation. The conformance with the ECOA Architecture Specification makes it possible to auto-generate the software required to integrate an instance of an ASC implementation onto an ECOA Platform, thereby ensuring that the ASC implementation is portable/reusable on any supported ECOA Platforms.

The ASC code should be developed such that it does not directly make use of any underlying functionality of the ECOA Software Platform, an exception is a Driver Component which may be platform dependent, refer to description in section 7.1.3.3.4. In addition, all the information required by an ASC should be contained within its context; as such the code should not make use of static variables (which could be re-used between ASC instances).

An ASC is a design entity, during the Component Development activity the ASC is implemented as one or more ECOA Module implementations, instances of these Module implementations are then deployed across the system by the System Integrator.

An ASC Service Operation is implemented as an ECOA Module Operation. Module Operations must also be used to specify communication between Modules of the same ASC. A Module instance can only communicate with another Module instance within the same ASC implementation using Module Operations which are invoked through the Container, and these communications will be specific (and unique) to each instance of that ASC.

An ECOA Module may be implemented from one or more code objects (e.g. subprograms), which may interact with each other directly.

An ECOA Container instance is software which can be auto-generated using the Component, Services, Data Types and Assembly specifications. It is responsible for mapping the ASC ECOA Module instances onto the underlying operating system or middleware. It is anticipated that tooling which generates the Container instance would be provided by the ECOA Platform supplier.

ECOA Module code can only communicate with Container code via the Container Interface and Module Interface [AS-Part 4]:

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

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

Figure 2, shows the relationships between the Component Development activities.

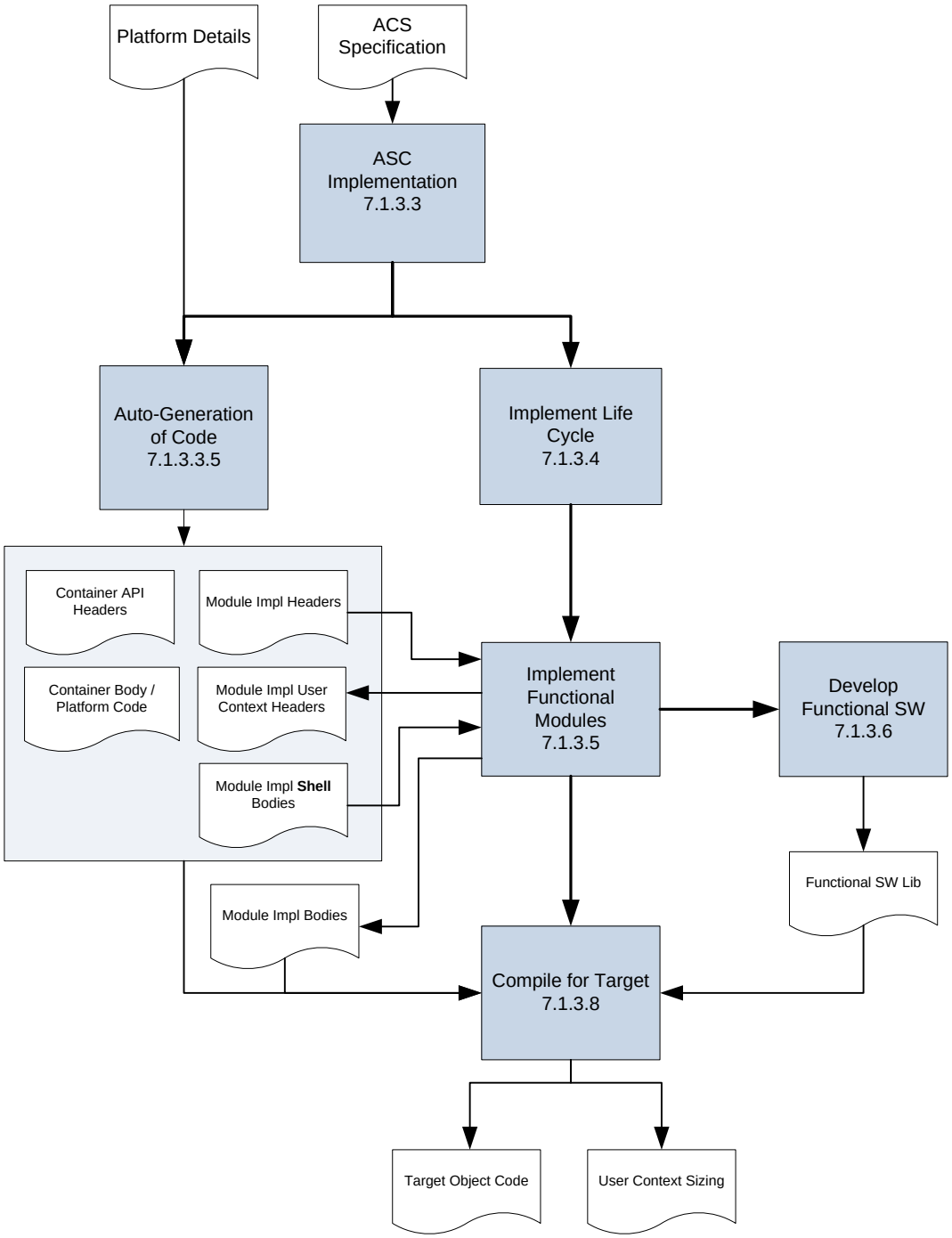

Figure 2– Component Developer Activities

7.1.3.2 Development Setup

The ECOA architecture specification mandates particular conventions for the naming of interfaces between the Container and Modules. As such the Developer will need to configure any development toolset they use to support ECOA Naming guidelines [AS-Part 4].

7.1.3.3 ASC Implementation

The ASC developer is responsible for developing the ECOA Module code in accordance with the Component specification, which specifies the Service behaviour and QoS, together with any relevant non-functional complementary requirements.

This activity should be supported by a Code Generator supplied by the ECOA Platform Provider. This tool generates the necessary code file stubs for the implementation, together with the required Container Interface and Module Interface.

ASC Service Operations are implemented as ECOA Module Operations. The delegation of Service Operations to a Module will be initially determined according to their functionality. Other considerations include: scheduling considerations, access to shared data, and potential re-use/sharing of the Operation by other ASCs.

An ASC must be implemented by at least one ECOA Module. Refer to [AS-Part 4] for a full description of the software interfaces between the Module and Container.

Outside of a Software Development Tool, the implementation of an ASC is specified by the “impl” XML file [AS-Part 7].

7.1.3.3.1 Operation Types

This section provides an overview of Service/Module Operation types, a detailed description of these Operations is provided in [AS-Part 3]:

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

- **Response Request**
  This Operation type is used to implement the equivalent of a remote procedure call. The request may carry data (‘in’ parameters) and the response may also carry data (‘out’ parameters).
  The client Module behaviour may be specified to either:
  - Synchronous – the requesting Module Operation thread waits for the response.
  - Asynchronous – the requesting Module Operation thread does not wait for a response, instead the response is handled using a callback Operation.

  Likewise, the server Module may send an immediate response, or a delayed response.

- **Versioned Data**
  This Operation type is used to share data between Modules (of the same ASC or different ASCs).

A requesting Module receives a copy of the data, however it should be noted that the data may not be synchronized with the latest update, particularly in a distributed system.

Optionally a notification can be provided to indicate that the data has been updated.

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

- **Dynamic Trigger**
  A Dynamic Trigger is used to provide an Event at a specified expiry time provided as an input Event. This is useful for implementing timeout behaviour following a specific event.

7.1.3.3.2 Tasking

ECOA Module Operations can only be invoked by Container threads. The operations are processed singularly in FIFO order. Module code is not permitted to use tasking Operations, including wait or delay type operations. A Container thread can invoke more than one Module, thereby providing support for multi-threaded behaviour.

Each Service Operation within an ASC may result in the execution of code in multiple Modules to achieve the desired functional behaviour. The flow of control through the various Modules to support a single Service Operation's behaviour is known as the Functional Chain. Each Service Operation has QoS attributes such as response time and minimal inter-arrival time which define their temporal performance requirements or guarantees.

Since the ECOA Module code cannot invoke any tasking Operations directly, the following paragraphs suggest possible implementations using ECOA operation types:

- **Periodic Operations**
  These operations can be implemented using Trigger Operations; these have a fixed duration assigned during integration.

- **Timed Behaviour** (e.g. delayed execution)
  These operations can be implemented using Dynamic Trigger Operations. These may have a variable duration assigned at execution time.

- **Multi-tasking**
  This is enabled either by:
  - Multiple Container threads each invoking a Module Instance,
  - A single Container thread invoking more than one Module Instance.

The developer should assign relative priorities to the ECOA Module and Trigger instances, these priorities will be taken into account by the System Integrator when scheduling the Container threads in order to ensure that the ASC instance can satisfy its QoS requirements.

All Module Operations are by default activating [AS-Part 3]. If an ASC developer requires periodic scheduling this can be implemented by specification of activating Trigger Operation(s) and assigning all other Operations on that module as non-activating. It should be noted that Module Lifecycle Operations are always activating.

7.1.3.3.3 Data Storage

The Versioned Data operation type [AS-Part 3] is useful for publishing data values to multiple recipients, which may be in the same partition, in different partitions, different processing nodes or even across ECOA Platforms. This is a push of the data to the Containers with readers, when a reader invokes the read operation it will receive the last value received by the container. A notifying event can be associated with Versioned Data, such that the Container of a Reader will generate an Event when a new value has been received. The associated "stamp" can be used by the Reader to determine staleness of the data.

The direct sharing of state data between ECOA Modules which could result in a module thread being blocked is not permitted. But the Versioned Data Module Operation is available for sharing state data, or alternatively state data could be managed by a single Module and accessed by other Modules using the Request Response Operation type.

See also PINFO in section 7.1.3.5.

Service Operations must be used to share state data between ASCs. The Component developer must elaborate his strategy for handling large data sets (i.e. Operation on a database ASC) in collaboration with the Software System Designer.

If there is a requirement to store the data offline then a Driver Component may be required to manage a data storage device, and provide Services via which other ASCs can have read and write access. See section 7.1.3.3.4 for more information on Driver Components.

Examples, including a database implementation, are provided on the ECOA website [WEB-1].

7.1.3.3.4 Driver Component

A Driver Component is an ASC which interfaces directly with a non-ECOA domain, examples of which include legacy software, a COTS software library or a hardware device. Non-ECOA software may include its own tasking which is outside of the tasking control provided by the ECOA Container instance.

Driver components are potentially less portable than other ECOA components having additional dependencies on the non-ECOA domain. This dependency of the Driver Component on additional non-ECOA software and hardware will need to be documented either in or referenced by the Component's Insertion Policy, see section 7.1.3.10.

Interfacing of a Driver Component with non-ECOA software is permitted provided the Component complies with its required provided and required QoS, and it does not block any Container Operation calls. Non-ECOA software may be interfaced through one or more ECOA Module instances which act as ECOA compatible façades or wrappers. These wrapper Modules may interact with the ECOA Infrastructure provided they do not block any container threads. The responsibility of the ECOA Module facades will be to map from ECOA-style execution to that expected by the non-ECOA software.

Refer to [AS-Part 3] for a full description of the mechanisms provided by ECOA for interfacing with non-ECOA domains.

7.1.3.3.5 Generation of Code Files

Following definition of the Modules, their Operations and inter Module links; code files are created for the following:

- Container API headers,
- Container Body/Platform Code,
- Module Implementation Headers,
- Module Implementation User Context Headers,
- Module Implementation code stubbed Bodies.

The creation of stubbed code files could be automated by the use of a Code Generator

The ASC Software Developer can then begin to enter code into the Module implementation stub bodies. This coding may create additional local subprograms to better manage/organize the code design.

It is anticipated that a Container Code Generator will support both the Reference ECOA Platform in addition to the target ECOA Platform. Initial code generation may be for a host platform (e.g. a Linux platform) to support test and incremental integration, before progressing to the actual target hardware.

7.1.3.4 Lifecycle Implementation

There are two levels of lifecycle to be considered:

- ECOA Module lifecycle,
- ASC Functional lifecycle.

7.1.3.4.1 ECOA Module lifecycle

The Module Lifecycle is managed by the ECOA Software Platform (module instantiation, fatal error handling); a full description is given by the AS Part-3, Mechanisms [AS-Part 3].

7.1.3.4.2 ASC Functional Lifecycle

Functional lifecycle of the ASC may depend on the whole system lifecycle and will therefore need to be compatible with System Design requirements. Those requirements will define when versioned data is initialized and what the ASC behaviour is when its required services are not functionally available.

It is anticipated that the Reference Architecture will specify the System functional life cycle where applicable.

The ASC developer will need to implement support for the functional lifecycle, which may require the handling of lifecycle management Services under direction of a hierarchy of Supervising/Managing ASCs, see guidance on System Management [WEB-1].

The developer must also manage any logging required to support System Integration. The logging requirement may, in part, be satisfied by the Container software.

7.1.3.5 Implement Functional Modules

The ASC developer is required to implement the code for the ECOA Module Operations. This implementation must not employ any OS calls, or language operations which could cause the

invoking Container thread to block. See section 7.1.3.6 for more guidance on software development of an ECOA Module.

Language Bindings are specified for the ECOA Module and Container APIs that facilitate communication between the Module Instances and their container. These are available for the following languages: C [AS-Part 8], C++ [AS-Part 9], Ada [AS-Part 10] and High Integrity Ada [AS-Part 11]. It is possible to implement an ASC using more than one coding language.

An ASC may be instantiated multiple times within a system; the ASC properties [AS-Part 3] provide a mechanism to tailor the behaviour of a particular instance. Similarly an ECOA Module may be instantiated multiple times and also has its own properties for tailoring the behaviour for each instance; a Module instance may access these ASC instance properties.

Given that a Module can be instantiated multiple times the developer must not make use of "global" state, and instead must use instance specific data via the Module Context. The Module context is described in further detail in [AS-Part 3]

ECOA specifies a mechanism, called PINFO [AS-Part 3] whereby Module instances can gain read-only access to data such as initialisation data. The Warm Start context [AS-Part 3] can be used to store state data but this is not persistent across a system shutdown. It should be noted that ECOA Platforms may differ in their support for PINFO, which could result in different latency and bandwidth behaviour for the reading of this data. If persistent data is required across power cycles, a non-ECOA mechanism must be used (e.g. by using a driver component).

7.1.3.6 Develop Functional Software

Functional Software includes algorithmic code such as navigation, math or graphics; its development can follow traditional software development methods, which may result in a number of compile-able items.

An ECOA Module can only be invoked by a single thread; therefore there are no issues with respect to data protection or thread synchronization to be addressed between Module Operations within a single Module's implementation code.

The data types required for module operation parameters must be manually or automatically generated from the ECOA datatypes, specified outside of a Software Development Tool in the ECOA "types" XML file [AS-Part 7]. The binding required for a software language is given in ECOA language bindings.

To further the reusability of the Module code, consideration should be given to developing the code in a configurable and generic manner, configuration can extend the reusability of the ASC as a whole and foster re-use of the code across more than one ASC. A Reference Architecture may recommend, or mandate, specific methods by which configuration or extend-ability of functionality should be achieved.

The implementation may utilize existing software libraries, however any dependencies on third party supplied libraries will need to be recorded, in the event of future reuse of the code. Third party libraries would either need to be supplied as part of the ECOA ASC implementation or the System Designer/Integrator must be informed of the dependency via the ASC Insertion Policy (see section 7.1.3.10) so that they can be acquired and linked appropriately.

It is recommended that the design and source code for the ECOA Modules and their decomposition code objects is managed by the developer, for instance, as part of a Functional Software Library. Such a library would encourage the reuse of the Modules in the future.

7.1.3.7 Fault Management

The developer will implement the required fault management behaviour on error occurrences as in any software development.

To report application-level errors, the developer can use raise_xxx_error API. If a module calls raise error, then it is reported to the ECOA Fault Handler.

It is anticipated that a Reference Architecture will provide guidance or specify rules such that all the ASCs integrated in the target System will exhibit the same behaviour in response to errors. This behaviour description could be supported by sequence diagrams.

In the case where the ASC is required to handle platform errors, the developer must implement the Fault Handler module [AS-Part 3].

The expected behaviour of the ECOA Fault Handler may be defined by the Reference Architecture. The recovery actions invoked by the ECOA Fault Handler include:

- Shutdown the faulty ASC,
- Restart the faulty ASC,
- Shutdown the faulty Protection Domain
- Restart the faulty Protection Domain

7.1.3.8 Compile Executables

As described earlier the ECOA Platform tooling should provide support for the generation of Container code files for the selected target platform.

The container code files together with the application code files will need to be compiled using a compiler which supports the target processor. Note: the generated binaries are specific to the selected target platform.

The building of an ECOA software system at this stage is essentially the same as for a traditional software system, see section 7.1.4 below.

7.1.3.9 Module Behaviour

A Module Behaviour allows characteristics of a Module Instance to be described, which may be used for deployment and early verification analysis (for example: consistency with the ASC level behaviour). It mainly gives a decomposition of Module treatments, allowing CPU resource needs to be assessed.

7.1.3.10 Insertion Policy

The developer should specify a Technical Insertion Policy, covering subjects such as:

- Assessment of required Stack/heap allocations.
  - Where practicable it is recommended that designs avoid using dynamic memory allocation and recursive subprogram calls as these can make such assessments more complex.

- Module properties
  - Why and how (they may be inherited from ASC properties).
- Calculation of Module Deadlines together with WCET.
- Assessment of Service/Operation QoS being provided.
- Assessment of Relative Module Priority from an ASC developer point of view.
- Processing platform requirements
  - Performance requirements relative to reference platform.
- RAM / EPROM/ PINFO/ Data Storage requirements.
- Software Library support required.
- Scheduling policy employed for the ASC verification.
- RTOS requirements
  - Where applicable.
- Assurance level
  - To which the ASC has been developed.
- Number of threads required.
- Reference Architecture compliance
  - Where applicable.
- For a Driver Component
  - Non-ECOA software / Hardware dependency description.

See also guidance on Technical Insertion Policy [WEB-1].

7.1.3.11 Component Validation

The ASC is validated against its requirements on the Test System. The test harness used to validate the ASC may potentially be reused on the Target System.

7.1.3.12 Outputs

The principal outputs of ASC development are:

- Target object code:
  - A binary per module or
  - A binary archive including all module binaries.
- Technical Insertion Policy:

  A Technical Insertion Policy is required unless the Component is only for internal use (not delivered); but then it is still recommended since it provides useful documentation and support for the ASC integration.
- Binary description (“bin-desc” XML file [AS-Part 7]).
- If required, a behavioural description for each module.
- If required, the updated Service QoS files, if during ASC development the QoS values have been refined.

In addition, the delivery shall include usual development documentation such as version description data, test report, etc.

7.1.4 Integration

The Integration phase has the following activities:

- Create Final Assembly,
- Create Deployment,
- Integrate on Target.

7.1.4.1 Create Final Assembly

This activity extends the Initial Assembly specification to include references to the component implementation; in addition any composites are required to be expanded to provide a flattened specification of the software system. The output of this activity is captured as the Final Assembly specification [AS-Part 7].

Outside of a Software Development Tool the Final Assembly is specified in the “impl_composite” XML file [AS-Part 7].

7.1.4.2 Create Deployment

This activity deploys the ASCs implementations across the Logical System. The deployment is captured in the “deployment” XML file [AS-Part 7], which using the ECOA Platform tooling is used to configure the ECOA Infrastructure to support the required communication links.

Deployment of ASC implementations across a system’s computing nodes has the same issues as those experienced in traditional software development with respect to communication bandwidth and latency, together with partitioning of safety and security critical software, see also Section 7.2.6.

The ECOA Module instances for each ASC instance are assigned to a Protection Domain which is in turn mapped onto a logical processor described by the Logical System. The Integrator needs to be aware of the possibility of name clashes, especially when integrating ASCs from different ASC Suppliers, for further details see Sections 8.1 & 8.3.

At present it is recommended that the Modules of an ASC instance are all deployed within the same Protection Domain. More than one ASC (from the same ASC Supplier) can be deployed within a Protection Domain. ASCs from different ASC Suppliers may be deployed in the same Protection Domain where the Module implementation names are known to be unique. Note that the ability to deploy ASCs spanning multiple PDs is a Platform Procurement option and therefore may not be supported on all platforms.

Data communication latency should be considered when deploying ECOA Modules. For example where an ECOA Module requires Service Operations from another Module which is deployed on a different computing node this could add significant latency into the functional chain, which could mean the Quality of Service cannot be satisfied.

In the case of Driver Components there may be dependencies to be satisfied at deployment, in particular with respect to hardware interfaces that might only be available on particular computing nodes.

The certification level of the ASC will also need to be considered, higher certification level ASC implementations will need to be partitioned from implementations developed according to a process tailored for lower criticality standards.

7.1.4.3 Integration on Target

This activity involves integration of the ECOA Modules, Containers, ECOA Platform specific software together with the OS/Middleware and Target hardware.

This is expected to follow the same approach to that taken for traditional software development. In particular, it is suggested that an iterative approach is taken for larger systems:

- integrating ECOA Module instances within a Protection Domain,
- integrating Protection Domains on a computing node,
- integrating computing nodes on an ECOA platform,
- And finally where multiple ECOA platforms comprise the system, these are integrated.

The build and deployment of the ECOA Module instances must follow the final Assembly / Deployment specifications, these specifications may require amendment during the integration dependent on issues encountered. Hardware and OS dependencies will need to be satisfied with the support of the ECOA Platform provider, including satisfying issues which are beyond the scope of ECOA, examples of which include System Time provision, hardware interface and OS configuration.

ECOA specific issues that will require addressing include the tuning of Module and Trigger instance container thread priorities for the target environment. The ASC developer will have suggested priorities for the ASC Module threads relative to others within the same ASC. However, the System Integrator will need to ensure that the deadlines for Modules from all the ASCs deployed can be satisfied.

In addition, any dependencies detailed in the ASC's Insertion Policy will need to be complied with, for example reservation of sufficient Stack and Heap space for the Module instances. The ASC developer will have designated depths for the Module instance queues (see section 7.2.2.4.4) but these depths could require tuning dependent on performance of the actual system.

Scheduling is performed by the Infrastructure, any scheduling policy supported by the OS/Middleware may be employed as required by the System Integrator. Scheduling analysis is outside the scope of ECOA; although it is anticipated that it would be carried out following existing, established methods.

The Integrator should validate that the system executes on the target system and that it does not exhaust the platform resources.

To improve accuracy of the early verification of future developments, performance information should be gathered and integrated into the toolset. This may include measured on target ECOA functional chain deadlines and platform communication latency times.

7.2 Software Platform Development

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

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

Figure 3 (following) illustrates a purely functional view of the elements which constitute an ECOA Software Platform.

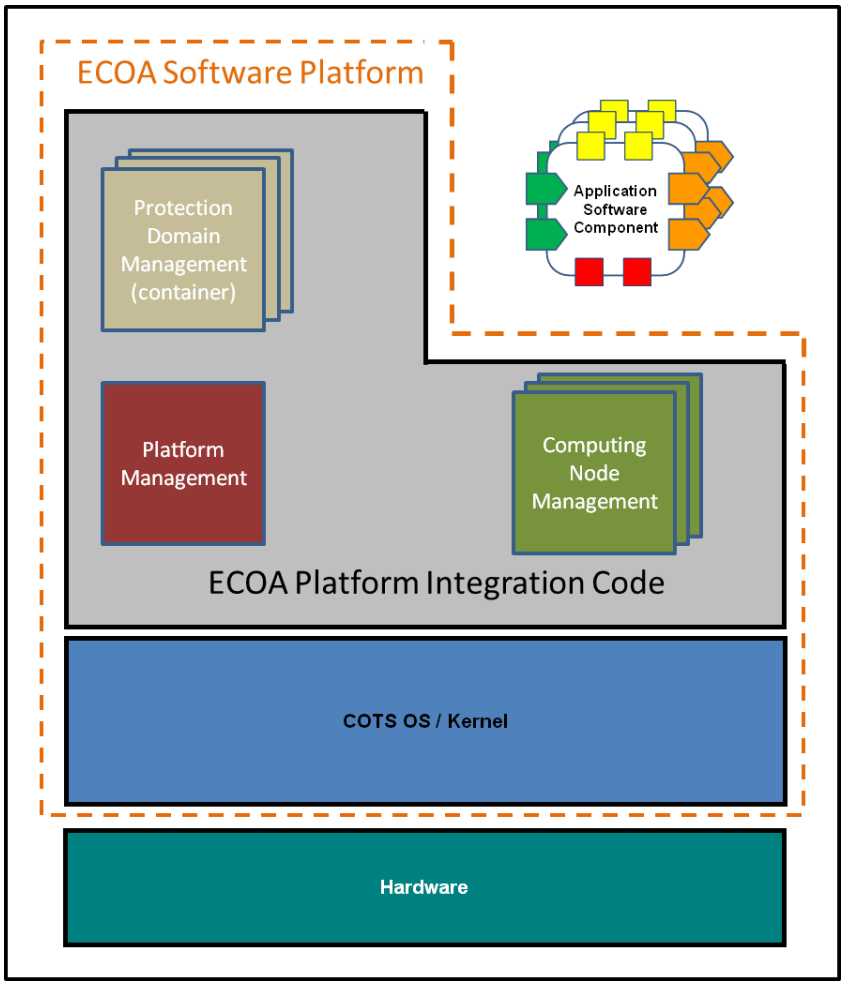

Figure 3 - Functional View of an ECOA Software Platform

The functional elements shown in Figure 3 can be mapped onto a deployment viewpoint

Figure 4 shows the Protection Domain Management functional element from a deployment viewpoint. Here, a single Protection Domain can provide functionality to manage the Protection Domain, alongside Container functionality to manage/support one or more ECOA Application Software Components. A Protection Domain provides spatial and potentially temporal partitioning.

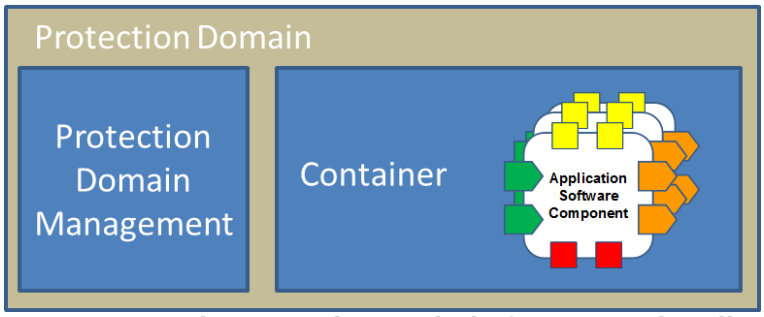

Figure 4 - Protection Domain Level Platform Functionality

In the same manner, Figure 5 shows the Computing Node Management functional element from a deployment viewpoint. Here, a single Computing Node can perform functionality to manage the node, along with one or more Protection Domains. The Computing Node Management functionality may be distributed between one or more Protection Domains, or be contained in a Protection Domain itself.

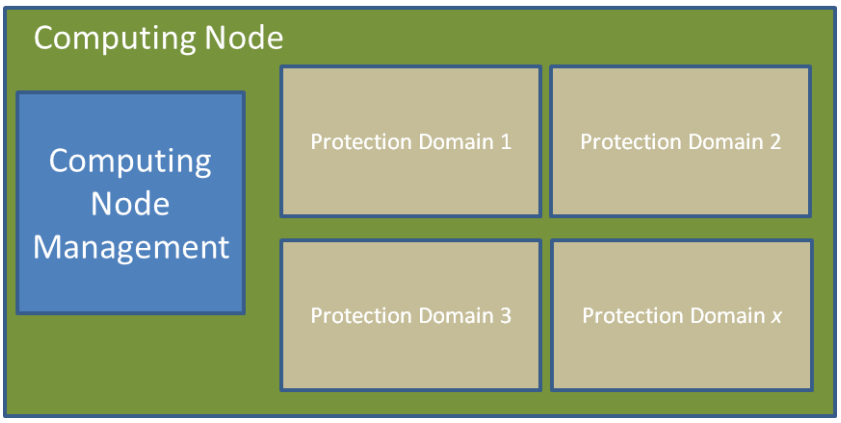

Figure 5 - Computing Node Level Platform Functionality

Finally, Figure 6 shows the Platform Management functional element from a deployment viewpoint. Here, a single Platform can perform functionality to manage the platform, alongside functionality to manage/support one or more Computing Nodes. The Platform Management functionality will necessarily be hosted on a Computing Node; however this may be distributed between the functional Computing Nodes, or be contained in a single Computing Node with other functions, or by itself.

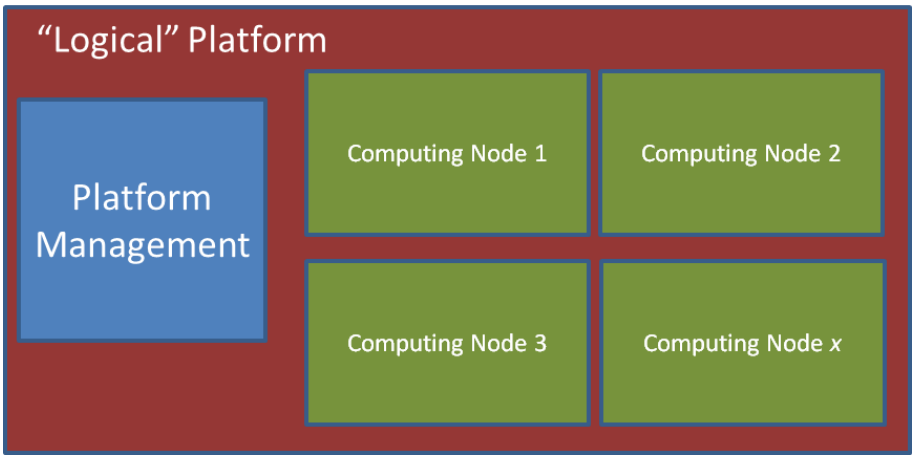

Figure 6 - Platform Level Platform Functionality

7.2.1 ECOA Software Platform Requirements

The requirements for an ECOA Software Platform are derived from the mechanism behaviour specified in the Mechanisms Reference Manual [AS-Part 3], and can be found in the Platform Requirements Reference Manual [AS-Part 5].

7.2.2 ECOA Software Platform Integration Code

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

Figure 8 shows a possible functional breakdown of the generic elements which are required, and Figure 9 shows the detail within the Component Implementation functional element.
Figure 7 gives a key to be used for these diagrams.

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

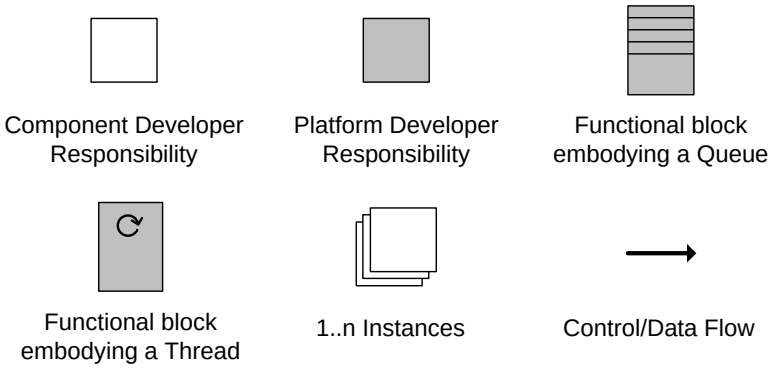

Figure 7 – Platform Integration Code Key to Diagrams

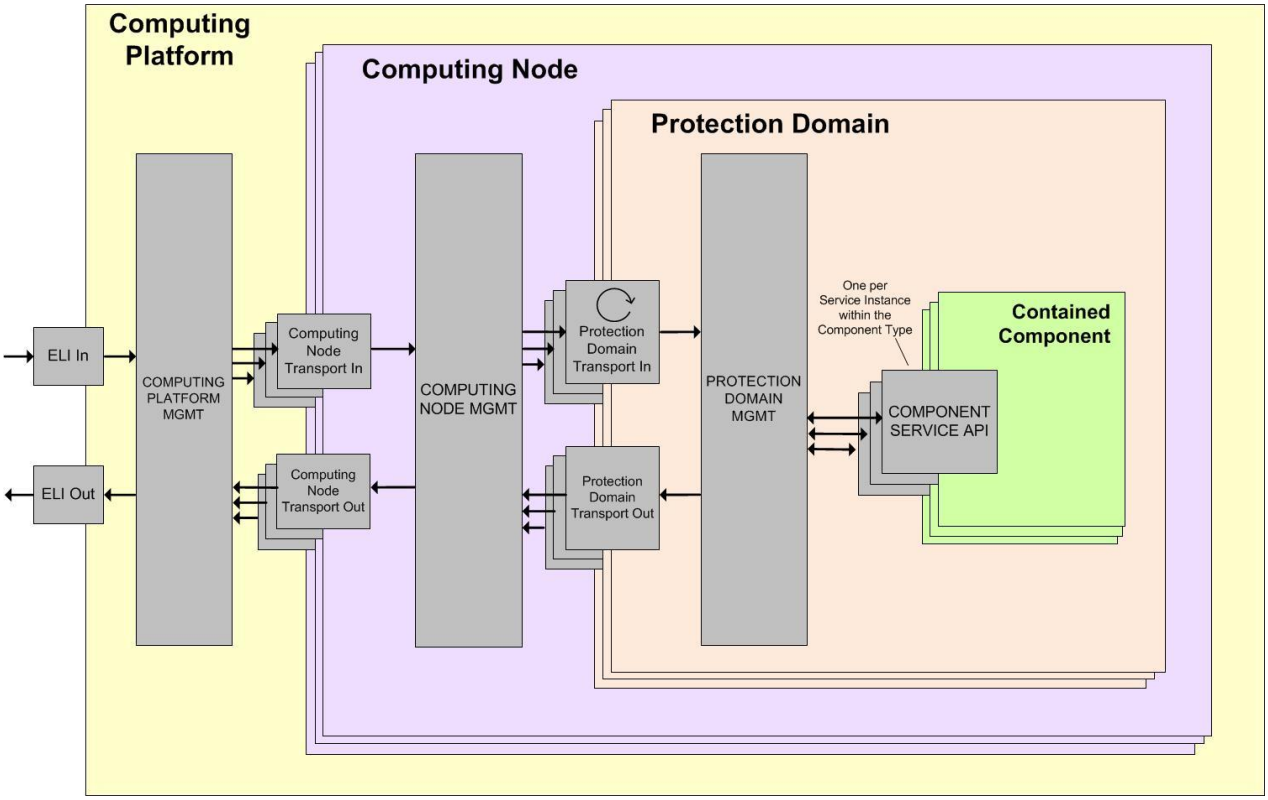

Figure 8 – Platform Integration Code Functional Design

7.2.2.1 Computing Platform

Within the Computing Platform boundary in Figure 8, the following functional elements are required:

7.2.2.1.1 Computing Platform Mgmt

The Computing Platform Mgmt is responsible for the management of the Computing Nodes within the same ECOA Platform, including power-up & initialization, fault management, configuration management and communications management. It routes operations between Computing Nodes within the same ECOA Platform and, if required, outside the ECOA Platform (using the ELI In/Out functionality). The method of routing is defined to maintain inter-operability; the ECOA Logical Interface (ELI) [AS-Part 6] must be used.

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

7.2.2.1.2 ELI In

The ELI In is responsible for receiving ELI messages directed to the ECOA Platform. The ELI message format is defined by the ECOA Specification [AS-Part 6]. The responsibilities of the ELI In can vary depending upon the transport medium used and the internal-platform communication methods. For example, if UDP is used, large ELI messages (greater than the maximum size of a UDP payload) are fragmented, so the ELI In would be responsible for reassembling a number of UDP packets into a complete ELI message.

7.2.2.1.3 ELI Out

The ELI Out is responsible for sending ELI messages to other ECOA Platforms. The ELI message format is defined by the ECOA Specification [AS-Part 6]. The responsibilities of the ELI Out can vary depending upon the transport medium used and the internal-platform communication methods. For example, if UDP is used, large ELI messages (greater than the maximum size of a UDP payload) will require fragmenting, and the ELI Out would be responsible for the fragmentation of the message.

7.2.2.2 Computing Node

Within the Computing Node boundary in Figure 8, in addition to the Protection Domains the following functional elements are required:

7.2.2.2.1 Computing Node Mgmt

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

7.2.2.2.2 Computing Node Transport In

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

7.2.2.2.3 Computing Node Transport Out

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

7.2.2.3 Protection Domain

Within the Protection Domain boundary in Figure 8, in addition to the Contained Component the following functional elements are required:

7.2.2.3.1 Protection Domain Mgmt

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

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

7.2.2.3.2 Protection Domain Transport In

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

7.2.2.3.3 Protection Domain Transport Out

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

7.2.2.4 Contained Component

Within the Contained Component boundary in Figure 8, the following functional elements are required, and shown in Figure 9. Note that for any Contained Component there may be many Module Implementations and there may be multiple instantiations (Module instances) of any given Module Implementation.

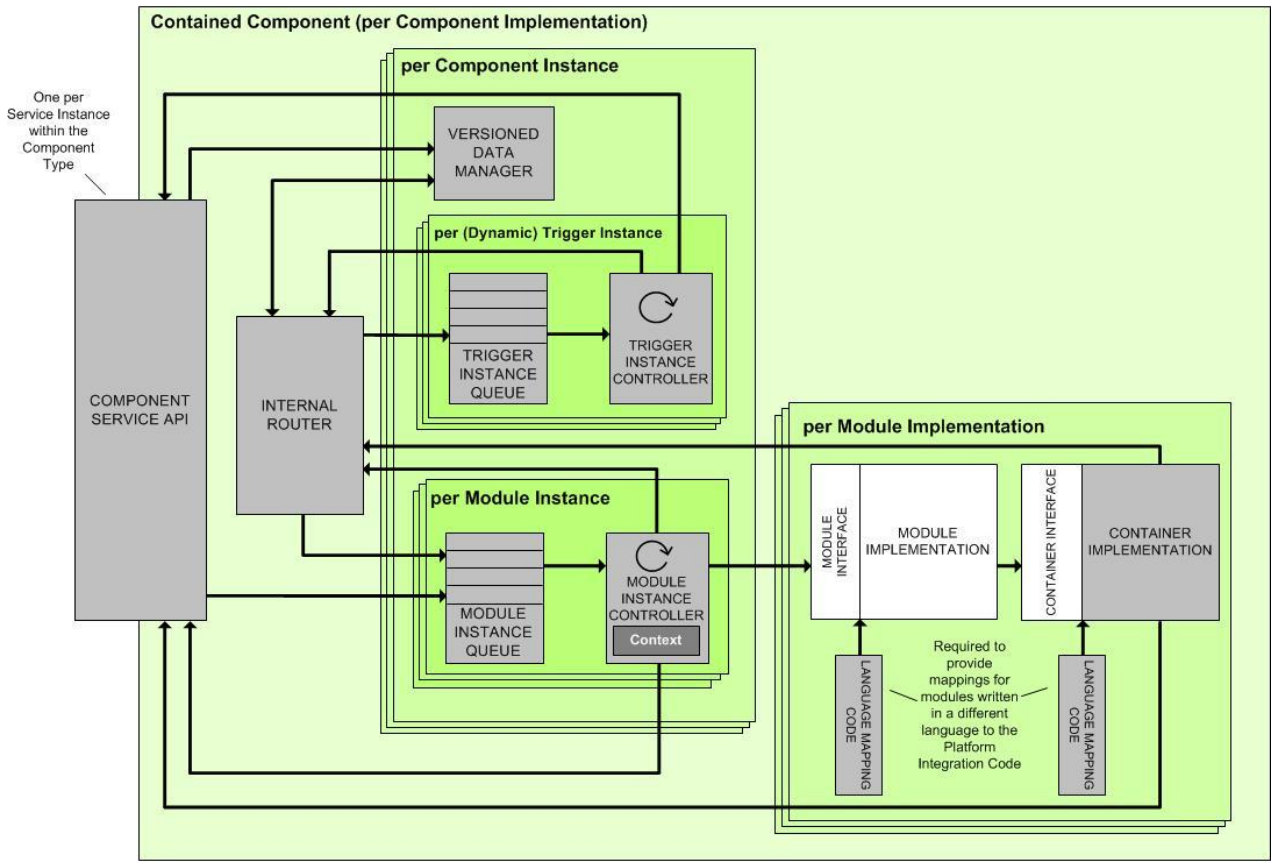

Figure 9 – Platform Integration Code Component Implementation

7.2.2.4.1 Module Implementation

The Module Implementation shown in Figure 8 represents the functional Module code provided by an ASC developer (typically delivered as a pre-compiled binary object). The functionality provided by the Module will be invoked by its associated Module Instance Controller using the ECOA defined Module Interface.

The Module Implementation code can also invoke Module Implementation specific Container operations, described in section 7.2.2.4.2 using the ECOA defined Module Implementation specific Container Interface.

7.2.2.4.2 Container Implementation

The Container Implementation comprises the code implementing the ECOA defined Container Interface for the specific Module Implementation, against which the ASC developer has compiled. The implementation of this interface must first identify the calling Module instance (typically by inspecting the Container private context) and then either invoke the Internal Router or ASC Service API (or both) depending upon the connectivity defined by the ASC implementation specification.

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

Note that the Internal Router is an abstract concept and may not necessarily exist at a technical level.

7.2.2.4.3 Module Instance Controller

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

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

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

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

7.2.2.4.4 Module Instance Queue

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

7.2.2.4.5 Trigger Instance Controller

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

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

7.2.2.4.6 Trigger Instance Queue

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

7.2.2.4.7 Internal Router

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

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

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

7.2.2.4.8 Versioned Data Manager

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

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

A more detailed description of Versioned Data Management is given in section 7.2.7.

7.2.2.4.9 Component Service API

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

7.2.2.4.9.1 Provided Services

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

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

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

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

7.2.2.4.9.2 Required Services

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

- Event sends (where the Event is "received by provider"),
- Request sends (async/sync).

  NOTE: for a synchronous call the module instance thread would be blocked by the Component Service API awaiting the response.

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

- Event received (where the Event is "sent by provider"),
- Response received,

- Versioned Data updates.

7.2.3 Internal Platform Communications

It is left to the ECOA Platform implementer to determine the optimal method of internal communications based upon the details of their ECOA Platform. The ECOA Platform is required to communicate with other ECOA Platforms using the ELI definition [AS-Part 6] to ensure interoperability.

7.2.4 Module Context

All Modules have an associated Module Context, which comprises information used by the ECOA Software Platform to manage the instances within the system, along with an optional user defined part that allows per instance state to be maintained. The user defined part of the Module Context is split into two further parts:

- User Context: contains instance specific data defined and managed by the module implementer. The platform is responsible for allocating the appropriate memory for this data to be stored.
- Warm Start Context: contains instance specific data defined and managed by the module implementer. The platform is responsible for ensuring this data is maintained between state transitions.

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

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

7.2.5 Scheduling

The low-level thread scheduling policy is not defined or mandated by ECOA, and is left to the System Integrator to define. Within an ASC implementation, the XML defines a relative priority assigned by the Component Implementer. This information can be used in the deployed system, within the deployment XML to assign a global priority for every Module instance and Trigger Instance. Additional information contained within insertion policies or module behaviour may also be used to aid the allocation of priorities.

7.2.6 Partitioning

Although ECOA does not mandate any particular partitioning scheme for ASCs, it provides the concept of Protection Domains for use where isolation in space and (potentially) time is required.

One partitioning model is to deploy a single ASC to each Protection Domain. This model could support mixing ASCs with different levels of assurance within the same system, and ultimately on the same Computing Node. Whether this is feasible depends upon the level of assurance being sought, and the nature of the safety requirement.

An alternate partitioning model is to deploy multiple ASCs within the same Protection Domain. This would usually be done for reasons of performance, or simplicity. Within the same Protection Domain the ECOA Platform would able to provide closely coupled communications paths, and hence provide a faster communication response between the ASCs, where required.

In addition, for rapid prototyping and deployment in non-critical environments it may be appropriate to deploy multiple ASCs within the same Protection Domain, as it will reduce the overheads of building a more complex distribution mechanism.

7.2.7 Versioned Data Management

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

Whilst dynamic allocation/de-allocation fulfils the requirements of the ECOA Software Platform and provides a useable capability to the Modules within the system, it may not be appropriate for all scenarios, as the temporal and spatial properties may be difficult to capture. In cases where it is desirable to have a statically allocated memory map, the ECOA Software Platform would be required to define a number of pre-allocated memory areas for use by Versioned Data Management. It is also possible to implement these mechanisms using fixed time algorithms to ensure there is no undesired variability in timing.

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

7.2.8 Module Lifecycle Management

The ECOA Mechanisms Reference Manual [AS-Part 3] describes the Runtime Lifecycle of Modules. The lifecycle is managed by the ECOA Software Platform.

It is the responsibility of the ECOA Software Platform to initialize and start all Modules (and Triggers) within each ASC. The order of startup of Modules, Triggers and Dynamic Triggers is specified in the ECOA Mechanisms Reference Manual [AS-Part 3].

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

7.2.9 Platform Level ELI

In order for multiple ECOA Platforms to coordinate their initialisation and start-up, a set of ECOA Platform level ELI messages have been defined, refer to [AS-Part 6]. These messages include notifications of changes to ECOA Platform status and Versioned Data Pulls. By using these messages and following the behaviours defined in Reference [AS-Part 3], an ECOA Platform can discover other ECOA Platforms, request the status of published Versioned Data, and be notified of any changes to these.

7.2.10 Persistent Information Management (PINFO)

ECOA provides a mechanism which allows Module Instances to access read-only persistent information whilst remaining portable. The platform is free to implement this capability in the most appropriate way for the deployment. For example, if a file system is available, the capability may be implemented using files. In other deployments, the capability may be implemented using non-volatile memory. The persistent information should remain available until it is explicitly deleted (i.e. state should be maintained between power cycles).

7.2.11 Fault Management

ECOA specifies a fault management concept which is split into two main areas. The first area is fault management at application level. This level of fault handling is not a concern for the platform provider as it is the responsibility of the Component Implementer.

The second area is the infrastructure level fault management capability. The platform provider is responsible for the implementation of this which broadly consists of:

- Detection and reporting of errors,
- Implementation of recovery action.

The ability to detect certain errors is likely to be platform dependent, meaning not all platforms will be able to detect all errors. For each ECOA Software Platform the set of errors that can be reported will need to be defined to allow the System Designer/System Integrator to define how they are managed.

It is the responsibility of the ECOA Software Platform to route the reporting of errors to the Fault Handler through the ECOA defined API.

Since the Fault Handler may be implemented either as a Component or as a separate piece of software invoked directly by the ECOA Software Platform, it is the responsibility of the ECOA Software Platform to invoke the Fault Handler in the appropriate way.

The implementation of the Fault Handler is the responsibility of the System Designer/System Integrator. It is responsible for receiving reported errors from the ECOA Software Platform and determining the most appropriate recovery action (assuming one is required). Once the decision has been made, the Fault Handler may invoke the recovery action through the API provided by the ECOA Software Platform.

The ECOA Software Platform is responsible for implementing the ECOA Recovery Action API. ECOA defines various possible recovery actions such as shutting down/restarting components, Protection Domains, computing nodes and platforms.

The implementation of the ECOA Software Platform may not implement all possible recovery actions. For each ECOA Software Platform the set of recovery actions it supports will need to be

defined to allow the System Designer/System Integrator to decide on the best recovery action for a particular scenario.

8 Legality Rules

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

8.1 Naming Rules and Conventions

When used within a Module, each of the above abstract APIs is instantiated at Module level per Operation. Each Operation declared in a Module will necessarily cause the definition of one or more subprograms used in that Module. As a consequence of the ECOA model, for system integration:

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

Note that where possible the ECOA XSD's enforce these legality rules.

8.2 Module Instances and Context

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

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

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

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

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

With OO languages, the Module will be instantiated as an object of a Module Implementation class declared by the user; its associated Container will be associated to an instance of an ECOA-generated Module Container class. The "User Module Context" will be declared within the user Module Implementation class as public attributes; all the infrastructure-level technical data will be declared by the ECOA-infrastructure within the corresponding (generated) Module Container class. In addition, the entry-points declared in the Container Interface are represented as methods of the Container object, so the Module instance object must have access to its corresponding Container object to enable it to call these methods. This is done by storing the Container reference as a public attribute in the Module Implementation class.

8.3 Guidance for Namespace Handling

Regarding naming conventions for ECOA ASCs, uniqueness of names should be checked:

- At Reference Architecture level (ASC names),
- At programme level (Module implementations names for new ASCs).

Workarounds in terms of names clashes include deploying an ASC in its own Protection Domain, or have one Protection Domain per ASC Supplier.

9 References

AS-Part 1	IAWG-ECOA-TR-001 / DGT 144474 Issue 6 Architecture Specification Part 1 – Concepts
AS-Part 2	IAWG-ECOA-TR-012 / DGT 144487 Issue 6 Architecture Specification Part 2 – Definitions
AS-Part 3	IAWG-ECOA-TR-007 / DGT 144482 Issue 6 Architecture Specification Part 3 – Mechanisms
AS-Part 4	IAWG-ECOA-TR-010 / DGT 144485 Issue 6 Architecture Specification Part 4 – Software Interface
AS-Part 5	IAWG-ECOA-TR-008 / DGT 144483 Issue 6 Architecture Specification Part 5 – High Level Platform Requirements
AS-Part 6	IAWG-ECOA-TR-006 / DGT 144481 Issue 6 Architecture Specification Part 6 – ECOA® Logical Interface
AS-Part 7	IAWG-ECOA-TR-011 / DGT 144486 Issue 6 Architecture Specification Part 7 – Metamodel
AS-Part 8	IAWG-ECOA-TR-004 / DGT 144477 Issue 6 Architecture Specification Part 8 – C Language Binding
AS-Part 9	IAWG-ECOA-TR-005 / DGT 144478 Issue 6 Architecture Specification Part 9 – C++ Language Binding
AS-Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 6 Architecture Specification Part 10 – Ada Language Binding
AS-Part 11	IAWG-ECOA-TR-031 / DGT 154934-A Issue 6 Architecture Specification Part 11: High Integrity Ada Language Binding
WEB-1	http://www.ecoa.technology/