



European Component Oriented Architecture (ECO A®) Collaboration Programme: ECO A Exploitation Guide

Prepared by Dassault Aviation

This document is prepared by Dassault Aviation and copyright is owned by Dassault Aviation. The information set out in this document is provided solely on an 'as is' basis and developers of this document make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.

1 Abbreviations and Acronyms

ACK	Acknowledge
API	Application Programming Interface
AS	Architecture Specification
ASC	Application Software Component
ASCTG	Application Software Components Test Generator
CAC	Clauses Administratives Communes « Armement »
CBSE	Component Based Software Engineering
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSMGVT	Connected System Model Generation and Verification Tool
DAL	Design Assurance Level
Db	Database
ECOА	European Component Oriented Architecture
EDT	ECOА Design Tool
ELI	ECOА Logical Interface
FAQ	Frequently Asked Questions
FR	France
HLR	High Level Requirements
HW	Hardware
IАWG	Industrial Avionics Working Group
IP	Intellectual Property or Internet Protocol
IT	Information Technology
LDP	Lightweight Development Platform tool
MMTI	Maritime Moving Target Indicator
MSCIGT	Module Skeleton and Container Interfaces Generator Tool
MW	Middleware
NA	Not Applicable
OS	Operating System
PC	Personal Computer
PINFO	Persistent Information (ECOА)
QoS	Quality of Service
SOA	Service Oriented Architecture
SW	Software
UML	Unified Modelling Language
WCET	Worst Case Execution Time
WP	Work Package
XML	eXtensible Markup Language
XSD	XML Schema Definition

2 References

1 ECOA Standard AS6
DEF STAN 00-973
RG AERO 000 973

2 ECOA website
www.ecoa.technology

3 Table of Contents

1	Abbreviations and Acronyms	2
2	References	3
3	Table of Contents	4
4	List of Figures	6
5	List of Tables	7
6	Introduction.....	8
6.1	What is ECOA?	8
6.2	What does the ECOA Standard consist in?	11
6.3	How to decide whether to choose ECOA?.....	11
7	Identification of Development Activities and Associated Tooling	14
7.1	Overview of Possible ECOA Tooling for the System Development Process	14
7.2	Overview of Available Open Source ECOA tooling.....	15
7.2.1	EDT – ECOA Editor	16
7.2.2	LDP – ECOA Engine	17
7.2.3	MSCIGT – ECOA Skeleton Generator	18
7.2.4	ASCTG – ECOA Test Generator	19
7.2.5	CSMGVT – ECOA Cork Generator	20
7.2.6	EXVT – ECOA Checker	21
7.3	About Other Tools.....	22
7.3.1	Remaining Possible Specific Tools	22
7.3.2	Available Non-Specific COTS Tools.....	22
7.4	Synthesis	23
8	Application Typologies.....	24
9	Roles	25
9.1	ECOA System Architect	27
9.2	ECOA System Integrator	29
9.3	ECOA ASC Supplier(s)	31
9.4	ECOA Compatible Platform Supplier.....	35
9.5	Illustration of the Relationship between Roles and Activities through the Component Development Workflow.....	37
10	Tutorial	38
10.1	Case of Study: Search and Rescue System.....	38
10.1.1	Logical Architecture	38
10.1.2	Functional Chains	40
10.1.3	Selection of a Subset of Components	41
10.2	Focus on Toolled Activities	42
10.2.1	UC1: As a System Architect, I want to define the applicative architecture of my system.....	43
10.2.2	UC2: As a System Architect, I want to modify an existing applicative architecture.....	45
10.2.3	UC3: As a System Integrator, I want to define components implementation architecture.....	47
10.2.4	UC4: As an ASC Supplier, I want to get an ECOA model specifically adapted to my components development	49
10.2.5	UC5: As an ASC Supplier, I want to generate source code for my component(s) modules	51
10.2.6	UC6 : As an ASC Supplier, I want to test one or several module(s) on the basis of my ECOA model and source code.....	53

10.2.7 UC7: As an ASC Supplier, I want to use binary files to integrate and test components	58
10.2.8 UC8: As a System Integrator, I want to define the software deployment onto hardware resources.....	59
10.2.9 UC9: As a Platform Provider, I want to provide an ECOA-compliant platform60	
10.3 Highlights on ECOA standard benefits	61
11 FAQ.....	62

4 List of Figures

Figure 1: System architecture based on ECOA Infrastructure	9
Figure 2: ECOA concepts go along with the continuity of system process development	11
Figure 3: Choice of a good perimeter for code refactoring with ECOA	12
Figure 4: Possible and existing specific tooling in the ECOA development process	14
Figure 5: Principle view of EDT	16
Figure 6: Basic toolchain for ECOA development process	23
Figure 7 : Application Typologies	24
Figure 8: Component development in a typology A application	37
Figure 9: Component development in a typology B application	37
Figure 10: Search & Rescue Logical Architecture	39
Figure 11: Tutorial components subset and its dependencies to other SAR components....	41
Figure 12: Use of tools in the development process.....	42
Figure 13: User Panel to launch ECOA tools	42
Figure 14: EDT New project creation (Tutorial UC1)	43
Figure 15: EDT Tree View of SAR example (Tutorial UC1)	44
Figure 16: EDT screenshot of a successful export (Tutorial UC1)	44
Figure 17: EDT export of the ECOA model (Tutorial UC1)	44
Figure 18: EDT screenshot of SAR example (Tutorial UC2).....	46
Figure 19: EDT - RTE implementation treeview (Tutorial UC2)	47
Figure 20: EDT screenshot of RTE implementation (Tutorial UC2)	48
Figure 21: The harness component replaces components connected to a subsystem	49
Figure 22: Expected harness to be generated by ASCTG (principle)	49
Figure 23: ASCTG configuration file (Tutorial UC4).....	50
Figure 24: Extract of HARNESS implementation file (Tutorial UC4)	50
Figure 25: Generated files (Tutorial UC4).....	50
Figure 26: New files created after using ASCTG and MSCIGT	51
Figure 27: Extract of RDD module source code (Tutorial UC5)	52
Figure 28: Creation of artefacts for functional tests with CSMGVT	53
Figure 29: Extract of RTE module source code	54
Figure 30: List of files created by CSMGVT (Tutorial UC6)	54
Figure 31: Extract of main code generated by CSMGVT (Tutorial UC6).....	55
Figure 32: Example of CSMGVT lifecycle logs (Tutorial UC6).....	55
Figure 33: Example of CSMGVT functional logs (Tutorial UC6)	55
Figure 34: Creation of artefacts for tests in an ECOA environment	56
Figure 35: Some files generated by LDP (Tutorial UC6).....	56
Figure 36: Generation of Protection Domains by LDP (Tutorial UC6).....	57
Figure 37: LDP execution dated logs about channels creation (Tutorial UC6).....	57

5 List of Tables

Table 1: ECOA AS6 Promotional Tools.....	15
Table 2: Definition of ECOA roles in the development process	25

6 Introduction

The ECOA exploitation guide is destined to a wide audience, and aims at helping to start with ECOA standard thanks to available open source and free tools.

It begins with a short introduction to ECOA concepts, with guidelines to help users in their decision to choose ECOA for their systems.

Then, the document reminds ECOA former tooling analysis, and explains the position of current available tools in the development process.

Before detailing activities performed by the different ECOA Business model roles, the document points the different typologies of applications the user may face, because they have an influence on roles perimeters. For each role and associated activities, the document defines applicable tools, and highlights the link with the ECOA approach (for example dependencies on ECOA model).

A tutorial section is provided in this version: the purpose is to offer a concrete step-by-step example of the ECOA development process, implementing tools, on a representative "Search and rescue" system.

To finish, the document provides with a FAQ section.

6.1 What is ECOA?

ECOA is an open standard that allows building a service-oriented architecture of ASCs (Application Software Components) which are independent of the underlying computing platform.

ECOA is designed to offer many properties:

- Combining a high level description of the applicative architecture (close to system considerations) with an implementation level describing deployed software artefacts (close to real-time considerations),
- Independence between functional code and technical infrastructure code,
- Portability of ASCs on any ECCPF (ECOA compliant computing platform),
- Interoperability between any ECCPFs,
- Compliance with any real-time scheduling policy.

ECOA allows implementing an ECOA Infrastructure, akin to a middleware, on top of any computing platform to host portable ASCs, regardless of its Operating System, as illustrated below.

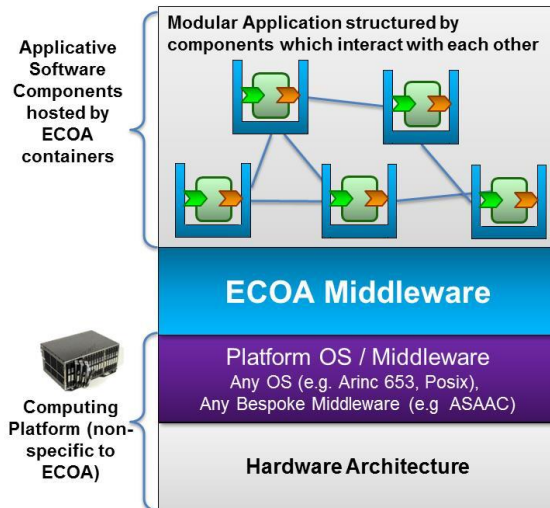


Figure 1: System architecture based on ECOA Infrastructure

Moreover:

- ECOA provides a technical solution for implementing software components in a collaborative way.
- The service oriented aspects of ECOA encourages application developers to focus on functional development rather than platform specific details. Consequently, the functional code becomes portable and easier to maintain.
- ECOA is a technology which caters for both new build and legacy upgrades. ECOA is capable of encompassing and interacting with legacy systems.

Key concepts:

This section gives a quick sum up of ECOA main concepts. For further details, please refer to ECOA Standard AS6 (see reference 1)

ECOA is build around the founding concept of **component**, to be considered as an applicative level artefact covering a system functional consistent sub-perimeter. The idea is to enable you to incrementally build your application by assembling components, each component bringing a new functionality. A component may be tailored to provide specific behaviour using **properties**.

In ECOA Business Model, the Application Software Component (ASC) is the unit of exchange between software developers and/or integrators.

To ensure components reusability and interoperability, ECOA defines an extensible standard for their exchange interfaces and software mechanisms they are allowed to use.

Exchange interfaces are based on the concept of **service** that a component can either provide or require. A service contains a set of related **operations**, each one relying on one of the three possible types of interaction:

- **Event** for sampling mode exchanges,
- **Versioned Data** for time-optimized sharing of last published data values,
- **Request-response** for transactions.

Thus, assembling components consists in connecting service links between provider components and consumer components. A components **assembly** defines an ECOA **application**.

This is the applicative architecture part of the standard.

This document is prepared by Dassault Aviation and copyright is owned by Dassault Aviation. The information set out in this document is provided solely on an 'as is' basis and developers of this document make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.

Then, for implementation purpose, ECOA contains a technical description level with complementary concepts:

A component is splitted into one or several **modules**, whose purpose is to bring together treatments having close real-time constraints and that will be sequentially executed. Modules identification is an essential step to define the technical real-time tasks that will be scheduled by the operating system.

Modules interfaces are defined using the same types of interactions than services operations at component level.

So, a **component implementation** describes the internal architecture of a component, made of modules, and links between either modules operations, or between a module operation and its associated service operation. This step also requires to define **relative priorities** between the modules belonging to the same component.

You need to know that, except for trigger-typed modules, a module is activated from the outside: as a matter of fact, in a module, treatments are associated to **entry-points**, which may be activated when the corresponding input exchange is received. This is called the **inversion-of-control** principle.

A really important concept associated to modules is the **container**. It consists in a technical glue code between modules specific API and the generic services of the ECOA platform. As this code is generated, applicative developers do not have to think about it. And yet, this is the key concept that allows ECOA to meet the need of both a standardized execution environment, and the independence from the underlying platform.

Once components implementations are defined, you are able to enrich the components in a **final assembly** by adding the selected implementation for each component, so that the technical artefacts of your application can be fully identified.

Then, the remaining major topic to define your system is to specify the deployment of software artefacts onto execution resources: in ECOA, you define a **deployment** by mapping modules, grouped within **protection domains** (representing software processes), onto a **logical system**. A logical system is a formal representation of significant platform characteristics for deployment, such as the identification of available nodes. The deployment specification requires to assign **absolute priorities** on modules, with respect to relative priorities already defined at component level.

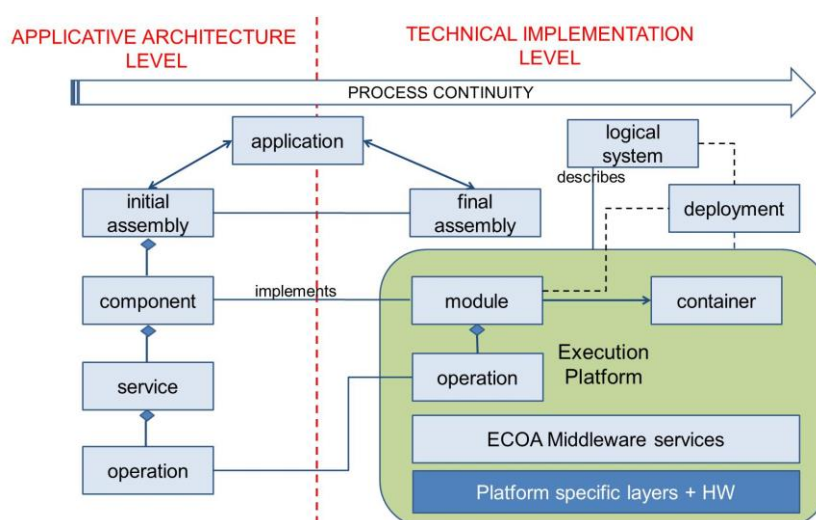


Figure 2: ECOA concepts go along with the continuity of system process development

6.2 What does the ECOA Standard consist in?

The ECOA standard is made of:

- Concepts and Mechanisms derived from CBSE and SOA,
- An XSD Meta-model that allows to specify a system using ECOA concepts:
 - At applicative architecture level: definition of components, services and associated exchanges types, assembly etc.,
 - At technical implementation level: definition of components implementation, deployment etc.,
- A Software interface and associated bindings for C, C++ and ADA languages,
- Platform requirements to claim ECOA compliance.

ECOA standard offers extensions possibilities, by defining new bindings for other languages or other applicative architecture description standards.



Remind that ECOA is not a methodology !

Using ECOA for your system design may give you tooling opportunities for some activities, but it does not impose technical choices and let you make your own system analysis.

6.3 How to decide whether to choose ECOA?

ECOA is the only architecture standard that brings the following two major benefits:

1. The **strength of implementation level concepts** (such as modules and containers), which offer powerful and perfectly adapted mechanisms for embedded real-time systems, as it can be proved by the use of ECOA in operational systems.
2. An **improvement of development process continuity** and components reuse, thanks to the integrated association of this implementation level with the applicative architecture level, which takes the form of a meta-standard compliant with lots of large public standards (gRPC, openAPI etc.) thanks to the definition of bindings.

Nevertheless, you have to consider several contextual topics before choosing to apply ECOA in your system development:

- Is it a new system development or a refactoring?

In case of a code refactoring, it is important to delimit a functional perimeter in your existing code that will facilitate the cohabitation with ECOA new developments. It is recommended to cover a functional perimeter that fits, as much as possible, with the frontiers of your previous code blocks.

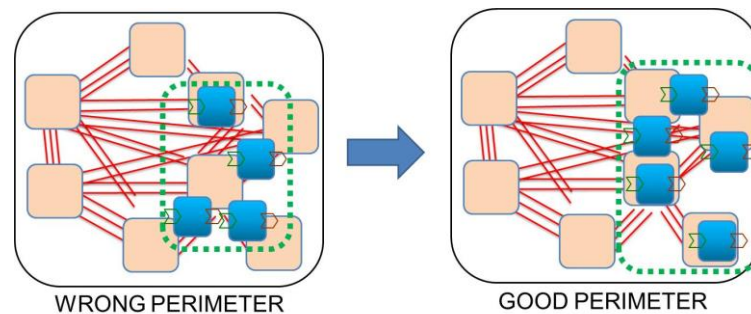


Figure 3: Choice of a good perimeter for code refactoring with ECOA

- Which other technologies do you intend to use?
 - Communication protocols,
 - Programmation languages
 Are they covered by the current ECOA standard or do you have to create new bindings?
- What are the characteristics of your execution platform regarding available resources, complexity of architecture?

ECOA implementation must be adapted regarding your application typology (described below), but in any case, ECOA allows you to easily manage different deployment strategies (knowing that ECOA is agnostic from your functional architecture and your scheduling policy, as far as the latter is consistent with your execution platform). This ability is particularly interesting for multi-nodes execution platforms, since ECOA automatically ensures deployment on nodes in accordance with XML files.
- Do you intend to reuse components in other applications or on other platforms?

Reusability is a strong argument to choose ECOA.
- Do you have certification requirements?

ECOA has been proved to be compliant with DO178C development processes, at least up to DAL C.
- What is the industrial organization of your system development?

In case of a multi-industrial partnership, a possible organization is described in the ECOA Business Model which clearly identifies roles and interactions between them. You can of course choose a different organization, or even work alone in your project context. What is important to have in mind is that, thanks to XML models, technical contracts that define software artefacts are non ambiguous, clear for all actors, which improves the quality of the development process.
- What is the relative cost of choosing ECOA considering your system complete lifecycle?

Adopting ECOA will require to:

 - Train your teams,

- Maybe replace some of your specific engineering tools with ECOA ones (to help you, some ECOA tools are available for free),
 - Maybe define complementary bindings,
 - Procure a middleware compliant with your execution platform requirements.
- In return, you will benefit from ECOA properties during your system lifecycle: tooling productivity gains, less errors thanks to process continuity, mastery of impacts in case of a component evolution, easy portability on other ECOA platforms etc.

Note that you can also refer to the FAQ section to get more elements for your decision.

7 Identification of Development Activities and Associated Tooling

7.1 Overview of Possible ECOA Tooling for the System Development Process

The following figure describes possible and existing specific tooling in the ECOA development process:

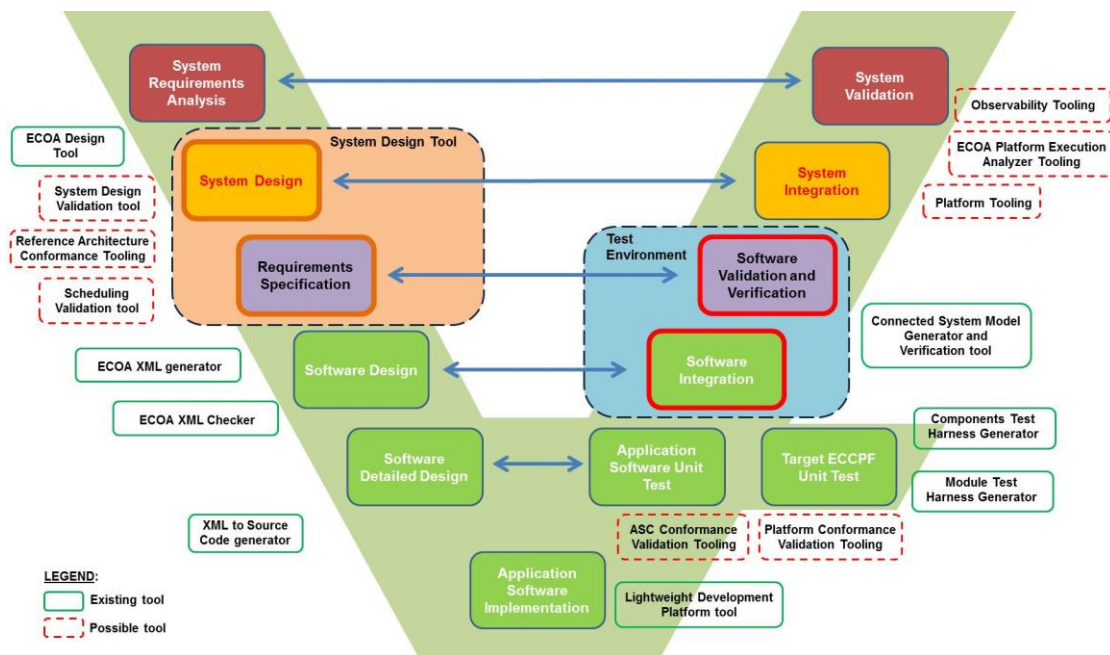


Figure 4: Possible and existing specific tooling in the ECOA development process

As you can see, lots of specific tools are already identified to enhance the development process thanks to the ECOA Standard, whose formal models make automatic analyses and generations possible.

Eight of these initial identified tools are now covered by six existing tools, which are available for free on ECOA website (see reference 2). You can custom them to your convenience as they are open source (respecting licence conditions).

7.2 Overview of Available Open Source ECOA tooling

To make the use of ECOA Standard easier, and to take then more rapidly advantage of its benefits, a set of six engineering tools is proposed, which allows to get a first evolutive ECOA design and development environment:





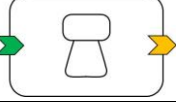

Official Name	Icon	Technical alias
ECOA EDITOR		EDT
ECOA ENGINE		LDP
ECOA SKELETON GENERATOR		MSCIGT
ECOA TEST GENERATOR		ASCTG
ECOA CORK GENERATOR		CSMGVT
ECOA CHECKER		EXVT

Table 1: ECOA AS6 Promotional Tools

This section gives a short description of each tool. You will find in the following sections of the document a more concrete approach, explaining how these tools can be used considering each role implied in the ECOA system development process.

Please note that all tools are compliant with ECOA Standard AS6.

7.2.1 EDT – ECOA Editor

ECO A Editor (also known as EDT meaning “ECO A Design Tool”) is a graphical editor allowing users to create step-by-step, or to simply update or visualize, an ECO A architecture.

It offers an XML import/export function allowing to easily switch from the graphical view to a corresponding consistent ECO A model, and to check imported models consistency.

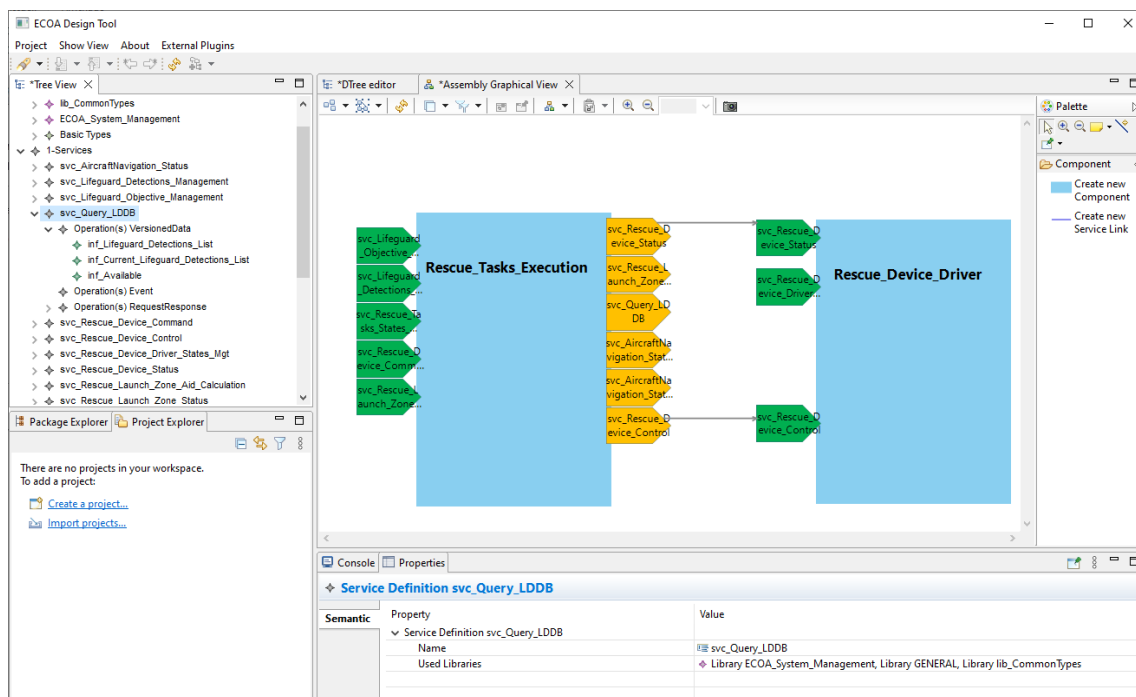


Figure 5: Principle view of EDT

Main benefits:

- No need to master ECOA meta-model semantic to use the standard,
- Easy appropriation of an ECOA system architecture thanks to user-friendly views.

Who is this tool for?

- Everybody.

What if I did not have such a tool?

- ECOA models would have to be manually written with risks of mistakes.
- An extensive reading of XML files would be necessary to get architectural views of components assembly and components implementation, which would require more effort to understand the system.

7.2.2 LDP – ECOA Engine

ECOA Engine (also known as LDP meaning “Lightweight Development Platform tool”) is a framework that requires a complete ECOA model, and source code for all the modules of described components. The tool is then able to generate an ECOA middleware that executes these components, following specified deployment rules. This middleware fully covers ECOA AS6 core specification, plus ELI, fault handling and graceful shutdown extensions. Only C and C++ bindings are available.

This middleware can be used for development, test or demonstration purpose, but is not designed for an embedded use.

Main benefits:

- Ability to execute an ECOA application on wide market means (COTS PC + Linux)
- Ability to generate binary files to provide partners with components to be executed in a compliant environment.

Who is this tool for?

- Everybody but mainly ECOA developers and integrators.

What if I did not have such a tool?

- Another ECOA environment would have to be found to execute components in a real-time context (it might be specific benches using embedded resources)
- Other means would have to be identified to generate binary deliveries (possibly manual compiling/linking directives), whenever you chose ECOA or not.

7.2.3 MSCIGT – ECOA Skeleton Generator

ECOA Skeleton Generator (also known as MSCIGT meaning “Module Skeleton and Container Interfaces Generator Tool”) is a tool that generates useful artefacts concerning ECOA modules implementation and test, such as:

- Source code headers and skeletons in accordance with ECOA API,
- Container source code,
- Partial module-level harness source code,
- Makefiles.

The tool only addresses C and C++ implementation languages.

Main benefits:

- Acceleration of ECOA modules development and test.

Who is this tool for?

- Developers.

What if I did not have such a tool?

- All previously presented artefacts would have to be manually written on the basis of ECOA bindings for C and C++ (with risks of mistakes),
- In case of evolution of components interfaces during the system development process, modifications would have to be manually done, taking care of consistency between files.
- Note that without ECOA, there is the same issue about source code production. MSCIGT offers a solution for ECOA.

7.2.4 ASCTG – ECOA Test Generator

From an ECOA model and a selection of components to be tested, ECOA Test Generator (also known as ASCTG meaning “Application Software Components Test Generator”) generates a new ECOA model describing both a new harness component and tested components. Each interface that is not initially connected within the set of components under test, is then automatically connected to the harness component in the assembly schema. The harness component is then able to stimulate and control all the external interfaces of the set of tested components (which is regarded as a subsystem).

The selection of components to be tested can be reduced to a single component.

Main benefits:

- Acceleration of components verification process.

Who is the tool for?

- Developers,
- Integrators.

What if I did not have such a tool?

- ECOA models would have to be manually updated to create component-level test artefacts, with risks of mistakes.
- This work would have to be repeated for each subsystem to be tested.
- Without ECOA, testing problematic would also have to be addressed, and solutions might be manual development (especially when there is no formal architecture model)

7.2.5 CSMGVT – ECOA Cork Generator

ECOA Cork Generator (also known as CSMGVT meaning “Connected System Model Generation and Verification Tool”) is a tool that allows a non real-time execution of ECOA components, apart from any ECOA middleware, by generating minimal stubs for each API service call. The main purpose of these stubs is to ensure communication between components. Time aspects are of course not significant.

The tool is only compliant with C and C++ implementation languages. It can be associated to a debugger tool such as gdb for functional code tuning.

Main benefits:

- Abstracting ECOA middleware to focus components verification on functional behavior,
- Compliance with office IT environment

Who is the tool for?

- Everybody.

What if I did not have such a tool?

- To focus on functional tests at component level, there would be no other solution than a manual development of stubbing artefacts, with risks of mistakes,
- In case of evolution of components interfaces during the system development process, modifications would have to be manually done, taking care of consistency between files.
- Without ECOA, functional testing problematic would also have to be addressed, and solutions might be manual development (especially when there is no formal architecture model).

7.2.6 EXVT – ECOA Checker

ECOA Checker (also known as EXVT meaning “ECOA XML Validation Tool”) is a tool that allows to check both the conformity of a set of XML files with ECOA AS6 specifications, and the consistency of described elements with each other. Note that EXVT can be executed on partial ECOA models: for example, you can choose to only check data types definition.

Main benefits:

- Ensuring the user to own a correct set of ECOA XML files

Who is the tool for?

- Everybody.

What if I did not have such a tool?

- An extensive reading of XML files would be necessary to get confident enough in an ECOA model validity, as remaining errors could introduce problems with different levels of severity in the system development process: misunderstanding between partners, integration issues due to inconsistent artefacts, unexpected behaviour during execution etc.
- This deep analysis, that requires a good knowledge of ECOA model legality rules, would have to be repeated (or partially repeated) each time the model is updated.

7.3 About Other Tools

7.3.1 Remaining Possible Specific Tools

According to Figure 4: Possible and existing specific tooling in the ECOA development process, there are some tools which were identified during previous ECOA phases. These tools fall into the following categories:

- Tools which would contribute to further promote ECOA to new adopters:
 - ASC Conformance Validation Tooling: test suite to check ASC definition and implementation according to ECOA rules.
 - Platform Conformance Validation Tooling: test suite to be executed on a platform to check the availability and correctness of ECOA API and software mechanisms.
- Tools which could improve productivity in the context of a given application on a specific target platform:
 - Scheduling Validation Tool: analysis of compliance between modules real-time properties and scheduling policy.
 - Platform tooling: for example, platform configuration management, components lifecycle management etc.
 - ECOA Platform Execution Analyzer Tooling: for example, performance measurements.
 - Observability Tooling: graphical tool allowing to activate and manage observability services in a compliant platform. It displays interpreted contents of selected operations parameters among components services.

Note that some of these tools may require extensions of the ECOA standard to specify complementary data (for example, real-time properties such as module WCET for scheduling analysis).

7.3.2 Available Non-Specific COTS Tools

It is of course possible and recommended to complete the ECOA engineering toolkit with non-specific COTS tools.

7.4 Synthesis

In the following sections, it will be considered that ECOA users have access to a basic toolchain containing:

- The set of ECOA specific open source tools,
- COTS tools: XML editor, compiler, linker and debug tools.

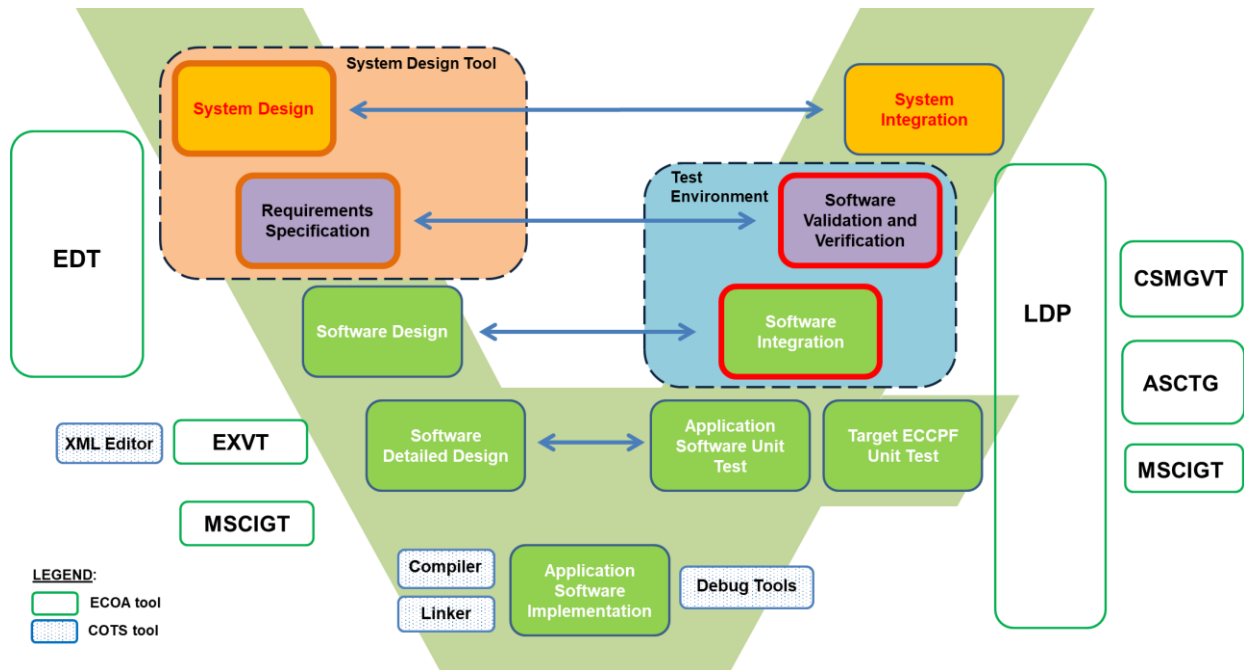


Figure 6: Basic toolchain for ECOA development process

8 Application Typologies

Two major application typologies are to be considered :

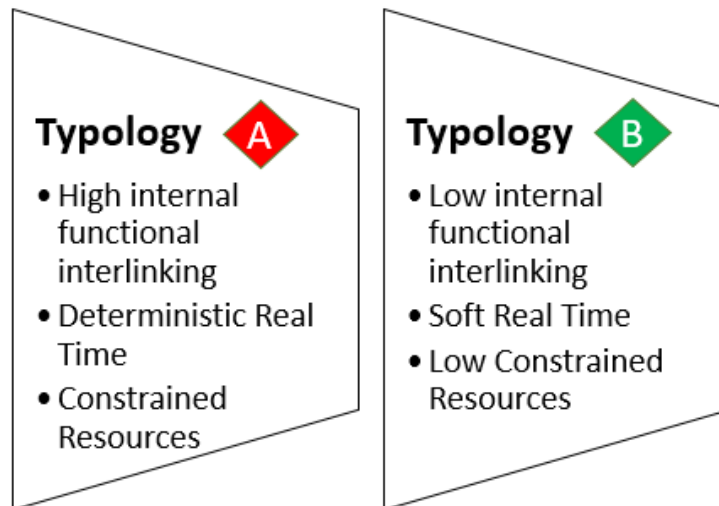


Figure 7 : Application Typologies

Application Typology **A** aggregates constraints as opposed to application typology **B** :

- Application typology **A** :
 - The infrastructure controls the execution of the functional code,
 - Data consistency at the module level to be managed (non-scalable if functionally or hierarchically managed at the component level),
 - Real – Time calculations,
 - A mastery of the splitting into modules of all the components is necessary for the consistency of data and performance on constrained resources. We will see in the following section that this activity for this given typology could be the responsibility of the ECOA System integrator.
- Application typology **B** :
 - Possibly relaxation of the inversion of control principle: this means that some applications may authorise the functional code to control its own execution,
 - Delayed calculations / Soft real-time calculations (response times are longer and the need for determinism is less critical),
 - High-performance and oversized resources (memory and computing power),
 - We will see in the following section that the mastery of the splitting into modules of each component for this given typology could be entrusted to the ECOA ASC Supplier.

9 Roles

Concerning the ECOA development process, the ECOA Business Model identifies contributing roles as shown in the table below:




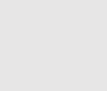




Role	Activity	V cycle	Definition
ECOAR® System Architect (System Designer)	System Architecture		The system designer is responsible for designing the system architecture captured in the ECOA assembly XML files as well as system level specification of ECOA components.
	Integrated System Validation		The system designer performs functional validation of ECOA components assembled together (i.e. functional validation of the integrated system).
ECOAR® System Integrator	Deployment Choice		The system integrator is responsible for the deployment of ECOA modules onto the CPU nodes of the target. This deployment is captured in the ECOA deployment XML files. In some cases, the system integrator is responsible for ECOA components breakdown into ECOA modules (see detailed tables below).
	Pre-Integration		The system integrator performs technical integration of ECOA components assembled together.
ECOAR® ASC Supplier(s)	Specifying Models		The ASC Supplier is responsible for detailed ECOA software component functional specification in compliance with system level specification, ECOA service contracts and insertion constraints. In some cases, the ASC Supplier is responsible for ECOA components breakdown into ECOA modules (see detailed tables below).
	Software Components		The ASC Supplier provides ECOA Software Component developed according to design rules. The ASC Supplier delivers Compiled code (binary).
ECOAR® Compatible Platform Supplier	ECOAR® Middleware		Platform Supplier provides the ECOA middleware for the execution platform, and tools for using this ECOA middleware on the target platform.
	Platform + OS		Platform Supplier provides Runtime Platform: <ul style="list-style-type: none"> ➤ HW = avionics platform ➤ Host Structure = OS (+ possible legacy middleware) ➤ ECOA Independent Platform Implementation Tools







Table 2: Definition of ECOA roles in the development process















For each role, the ECOA Business Model also precisely defines responsibilities, from which main activities are extracted and presented in this section.

For each role, the following arrays define activities attached to each ECOA role. This is how these arrays must be interpreted:








- Column “Activity” identifies an activity.
- Column “Sub-activity” refines this activity into sub-activities.
- Column “Application typology” indicates how application typologies may impact the activity, sometimes even determining whether the activity is applicable.
- Column “Available tools” specifies which tools you can use to help you performing the activity.
- Column “Relation to ECOA” explains how the activity interfere with ECOA Standard, e.g. by enriching the ECOA system model or by using it, by relying on ECOA mechanisms etc.
- The next column lists some possible ECOA inputs to perform an activity that is not directly addressed by ECOA.
- The last column refers to helpers that users can apply to get interesting guidelines for their activities.














9.1 ECOA System Architect














Activity	Sub-activity	Application typology	Available tools	Relation with ECOA	Possible ECOA inputs to achieve the activity	Helpers
1. ECOA Platform selection						
	Platform requirements identification	may be a requirement		Required ECOA perimeter must be defined	ECOA core and extensions in AS6	ECOA Developers Guide available on the ECOA Website (see reference 2)
	Analysis of procurement alternatives	may be a consequence				
	Logical system definition			Specified in the ECOA model as an output		
2. Functional chains definition		 				MBSE process (e.g relying on languages such as UML or SySML)
3. Identification of applicable system architecture rules		 		Might impact ECOA components reusability		Micro-services approach
4. Strategy selection for integration to non-ECOA domains		 		Refers to ECOA mechanisms	Driver components ELI protocol	

5. Iterative definition of applicative architecture						
	Components / services / Data types identification including QoS	 	EDT	Specified in the ECOA model as an output		ECOA Developers Guide available on the ECOA Website (see reference 2)
	Components system specification			Refers to ECOA components interfaces		
	Assembly definition	 	EDT	Specified in the ECOA model as an output		ECOA Developers Guide available on the ECOA Website (see reference 2)
	Early verification	 		Partially relies on ECOA model attributes	QoS attributes	
6. Functional validation of the application						
	Components test definition	 				
	Non-real-time verification of functional chains	 	CSMGVT			
	Verification of integrated components on target platform	 		Might benefit from observability tools based on ECOA	Exchange Ids Data types definition	
7. System demonstration on lightweight means		 	LDP	If LDP covers ECOA required perimeter		








9.2 ECOA System Integrator














Activity	Sub-activity	Application typology	Available tools	Relation with ECOA	Possible ECOA inputs to achieve the activity	Helpers
1. Definition of rules related to components technical behaviour		 		In accordance with ECOA mechanisms		
2. Component Design						
	Component breakdown into modules and treatments mapping		EDT	Specified in the ECOA model as an output	Operations QoS in component services	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
	Time allocation for components modules				Logical system	
	Insertion constraints	Stronger for  				
	Modules activation policy and priorities		EDT	Specified in the ECOA model (operation attribute "activating") as an output	Operations QoS in component services	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)





	Modules exchanges end FIFO sizing		EDT	Specified in the ECOA model as an output	Data types definition	
	Initialization procedures			In accordance with ECOA lifecycle mechanisms	Components and Modules lifecycles	
	Fault management policy			In accordance with ECOA fault handling mechanisms	Fault Handler	
	Context management and content	 		In accordance with ECOA software mechanisms	Warm start context	
	3. Creation of final assembly	 		Specified in the ECOA model as an output	Components Implementations Initial assembly	
	4. Deployment of components modules onto platform resources	 				
	Choice of scheduling policy	 		No ECOA specificity		ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
	Definition of Protection Domains	 		Specified in the ECOA model as an output		

	Definition of Modules absolute priorities	 		Specified in the ECOA model as an output	Module relative priorities within components	
	Schedulability analysis	 		Partially relies on ECOA model attributes	Logical system Deployment attributes	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
5. Test of delivered components			ASCTG + LDP + Debug Tools	API from ECOA binding for chosen language	Components Definition Deployment	
6. Technical integration of components						
	Within a protection domain	 	LDP	If LDP covers ECOA required perimeter	Deployment	
	Protection domains on a computing node	 	LDP	If LDP covers ECOA required perimeter	Deployment	
	Computing nodes on a platform	 	LDP	If LDP covers ECOA required perimeter	Deployment	
	Multi-platforms integration	 	LDP	If LDP covers ECOA required perimeter	Deployment	









9.3 ECOA ASC Supplier(s)



Activity	Sub-activity	Application typology	Available tools	Relation with ECOA	Possible ECOA inputs to achieve the activity	Helpers
1. Component Design						
	Component breakdown into modules and treatments mapping (including HLR definition)		EDT	Modules specified in the ECOA model as an output	Operations QoS in component services	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
	Modules activation policy and priorities		EDT	Specified in the ECOA model (operation attribute "activating") as an output	Operations QoS in component services	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
	Modules exchanges FIFO sizing		EDT	Specified in the ECOA model as an output	Data types definition	
	Initialization procedures			In accordance with ECOA lifecycle mechanisms	Components and Modules lifecycles	
	Fault management policy			In accordance with ECOA fault handling mechanisms	Fault Handler	
	Context management and content	 		In accordance with ECOA software mechanisms	Warm start context	
2. Developing the component						

	Types libraries	 	MSCIGT	API from ECOA binding for chosen language		
	External libraries	 		No ECOA specificity		
	Module source code	 	MSCIGT	API from ECOA binding for chosen language		
	Container source code	 	MSCIGT	API from ECOA binding for chosen language		
3. Conducting stand-alone verification activities						Applicable process for the expected quality assurance level (independent of ECOA)
	Module level testing	 	MSCIGT + LDP + debug tools	API from ECOA binding for chosen language	Component implementation	
	Component level testing		ASCTG + MSCIGT + LDP + debug tools	API from ECOA binding for chosen language Deployment specified in the ECOA model	Component Definition	ECOA Guidance for rhythmic models available on the ECOA Website (see reference 2)
	Functional test at component level	 	CSMGVT + debug tools	API from ECOA binding for chosen language	Component Definition	
4. Providing the associated artefacts						

	Building artefacts	 	Compiler + Linker	No ECOA specificity		
	Binary files characteristics	 		Specified in the ECOA model (bin-desc.xml)		

9.4 ECOA Compatible Platform Supplier

Activity	Sub-activity	Application typology	Available tools	Relation with ECOA	Possible ECOA inputs to achieve the activity	Helpers
1. Platform Design		 		ECOA model (logical system definition) is an output		ECOA Developers Guide available on the ECOA Website (see reference 2)
2. Developing an ECOA Middleware		 				
	ECOA core development		MSCIGT	conformity to AS6	ECOA standard	LDP source code
	ECOA extensions development (AS6 compliant)			conformity to AS6	ECOA AS6 optional parts	
3. Specific additional features		 				
	New Extensions (example: new binding, fine grain deployment...)					
	Tooling development (observability...)				ECOA model (services definition)	
4. Integration on Hardware		 				

5. Validation and certification		 				
	ECOA conformity tests		LDP	conformity to AS6	ECOA standard	ECOA examples available on the ECOA Website (see reference 2)
	Certification process					

9.5 Illustration of the Relationship between Roles and Activities through the Component Development Workflow

This section aims at illustrating through the component development workflow the relationships between role and activities while taking into account the application typology.

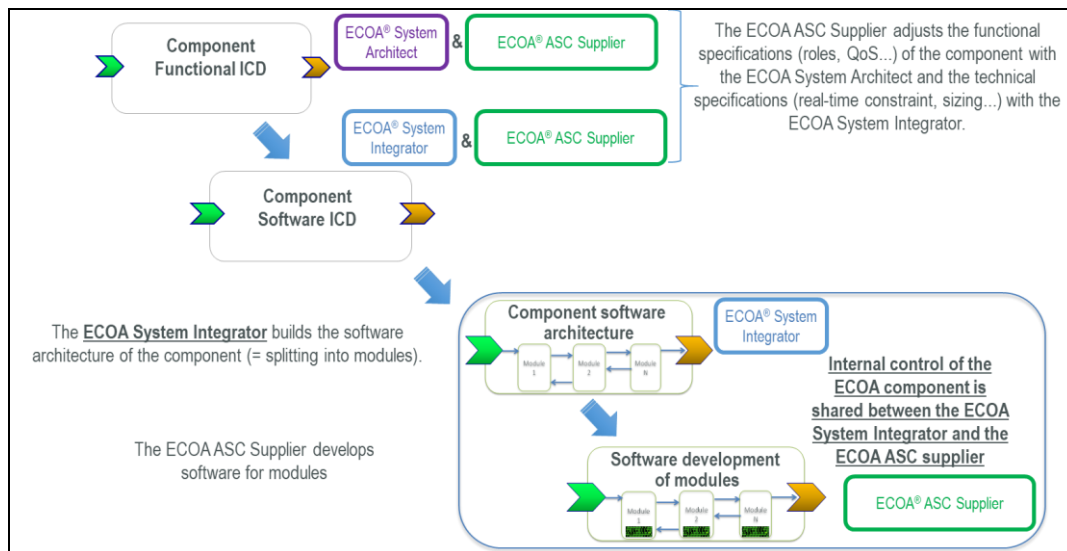


Figure 8: Component development in a typology A application

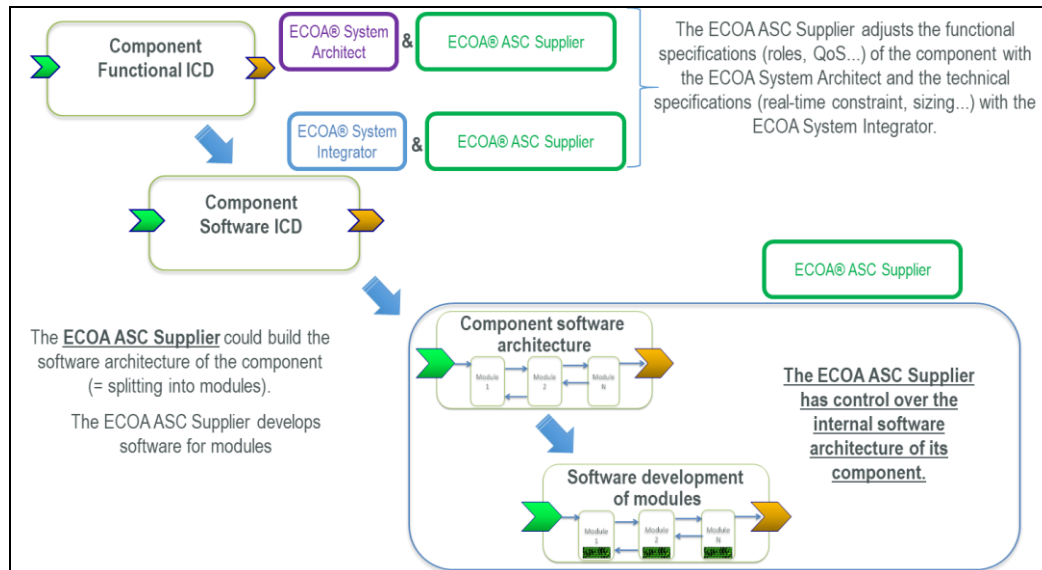


Figure 9: Component development in a typology B application

10 Tutorial

10.1 Case of Study: Search and Rescue System

10.1.1 Logical Architecture

The tutorial whose purpose is to explain the differences and benefits from using ECOA is based on a case of study: Search and Rescue system.

This case of study is a representation of a rescue aircraft and a ground control station working together to perform a search and rescue mission within the defined emergency area.

The rescue aircraft will fly a route whilst prosecuting a set of several identified potential objective for rescue under control of a ground control station operator (through HMI).

In addition it will be possible to detect additional Maritime Moving Target Indicators. Upon detection of the MMTIs these will be displayed to the operator to allow them to be selected as an item of interest.

Once selected as an item of interest sensor video will be displayed to the operator. The operator may switch to a radar map image to determine if the item should be designated as an objective for rescue.

Once designated as an objective for rescue it will be possible for the operator to select it as the primary objective for that stage of the mission and to order to the aircraft the rescue sequence.

The following figure describes its logical architecture.

The Search and Rescue system implements several functional chains that will be discussed in detail in the following section:

- Prepare rescue device,
- Display MMTI radar detections and add them to lifeguard detection Db,
- Select objective for sensors aiming,
- Perform radar map acquisition of objective for sensors aiming,
- Perform optical tracking of objective for sensors aiming,
- Select MMTI as potential objective for rescue and tag it in Db,
- Select confirmed rescue objective among potential rescue objectives,
- Launch aid computed,
- Display launch aid cue,
- Rescue drop sequence.

Please note that ECOA artefacts related to this tutorial (XML files and source code) will be available on the ECOA website (SAR-example).

In the rest of the document, it is considered that elements are created in accordance with this given SAR-example.



Figure 10: Search & Rescue Logical Architecture

10.1.2 Functional Chains

- **Prepare rescue device:** When the operator launches the preparation of the rescue device, a request is sent to the “Rescue_Controller” component, which sends an “inflate” event to the “Rescue_Tasks_Execution”. This event is transmitted to the rescue device driver, the rescue device is prepared and the rescue device driver writes a version data indicating the new preparation status of the rescue device. This data is read by the “Rescue_Controller” and this component replies to the “Operator_Station_Manager” that the rescue device is now ready. This information is displayed to the operator.
- **Display MMTI radar detections and add them to lifeguard detection Db:** Newly detected MMTI radar detections are added to the “Lifeguard_Detections_Db” by the “Lifeguard_Search_Execution”. They are also sent to the “Lifeguard_Search_Controller” and then sent to the “Operator_Station_Manager”.
- **Select objective for sensors aiming:** When the operator selects a lifeguard detection, the “Operator_Station_Manager” sends a request to the “Lifeguard_Search_Controller” to aim the optical sensor at it. If the “Lifeguard_Search_Controller” was in tracking mode, the “Lifeguard_Search_Controller” validates the tracking request and sends an event to the “Rescue_Mission_Manager” to request activation of high rate state vector. The “Lifeguard_Search_Controller” sends an event to order the “Lifeguard_Search_Execution” to aim the optical sensor. The “Lifeguard_Search_Execution” then aims the optical sensor at the intended objective position (which is the position of the lifeguard detection). Next, the optical sensor sends a video stream parameter and will send periodically video stream buffers with one event per buffer. The “Lifeguard_Search_Execution” will receive those buffers and will send an image flow to the “Operator_Station_Manager”.
- **Perform radar map acquisition of objective for sensors aiming:** When the operator requests a radar map, request passes through the “Operator_Station_Manager”, the “Lifeguard_Search_Execution” and then reaches the Radar sensor. The “Lifeguard_Search_Execution” sets objective position so as to aim the radar map center. When the radar has performed map acquisition, it is being displayed to the operator.
- **Perform optical tracking of objective for sensors aiming:** When the operator switches to optical tracking mode, a request is sent by the “Operator_Station_Manager”, passes through the “Lifeguard_Search_Controller” and reaches the “Lifeguard_Search_Execution”. The “Lifeguard_Search_Execution” then configures the optical sensor and sends periodically an updated tracked position. The video parameter is sent by the optical sensor to the “Lifeguard_Search_Execution”. Video is finally displayed by the “Operator_Station_Manager”.
- **Select MMTI as potential objective for rescue and tag it in Db:** When the operator selects a MMTI track as potential objective, the “Operator_Station_Manager” sends a request that passes through the “Rescue_Controller” and reaches the “Rescue_Tasks_Execution”. The “Rescue_Tasks_Execution” is then responsible to tag this lifeguard detection as potential objective in the “Lifeguard_Detections_Db”. The “Rescue_Tasks_Execution” can then read objectives list from the

“Lifeguard_Detections_Db” and pass it to the “Operator_Station_Manager” via the “Rescue_Controller”.

- **Select confirmed rescue objective among potential rescue objectives:** When the operator confirms a potential rescue objective, a request passes through the “Rescue_Controller” and reaches the “Rescue_Tasks_Execution”. The “Rescue_Tasks_Execution” downloads the objective position into the “Rescue_Device_Driver”. Downloaded position is then displayed to the operator.
- **Launch aid computed:** When a new air vehicle position is provided, the “Rescue_Tasks_Execution” requests a new launch aid calculation. A rescue device drop zone is returned and the “Rescue_Tasks_Execution” publishes the rescue device basket to the “Operator_Station_Manager” through the “Rescue_Controller”.
- **Display launch aid cue:** When a new rescue device launch aid calculation is performed, the “Rescue_Tasks_Execution” will provide to the “Operator_Station_Manager” through the “Rescue_Controller” a drop cue if the the air vehicle is within the rescue device drop zone.
- **Rescue drop sequence:** When the operator pushes the DROP button, the “Operator_Station_Manager” will send to the “Rescue_Controller” a rescue device drop command. The “Rescue_Controller” will then check the rescue device preparation status, loaded objective position and drop cue. If verification is successful a rescue device drop command will be sent to the rescue device driver.

10.1.3 Selection of a Subset of Components

The tutorial will focus on development activities for a subset of Search and Rescue system (SAR) composed of “Rescue_Device_Driver” and “Rescue_Tasks_Execution” components. This selection has been driven by the large combination of exchanges managed by these components.

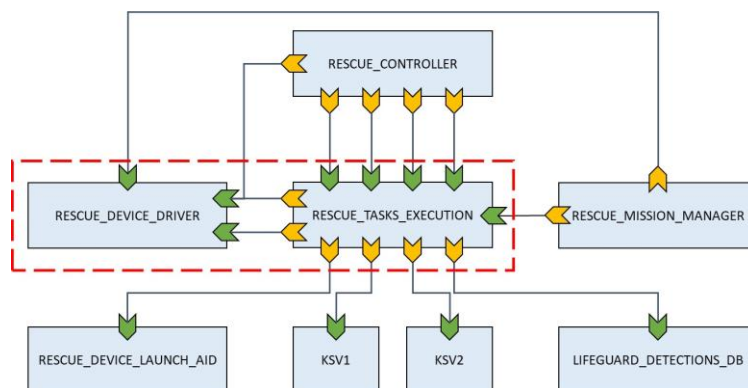


Figure 11: Tutorial components subset and its dependencies to other SAR components

10.2 Focus on Toolled Activities

This is a step-by-step presentation of activities performed using available tools.

Here is an overview of a possible process based on given tools to develop an ECOA application. At each step, EXVT can be used to check an ECOA input model.

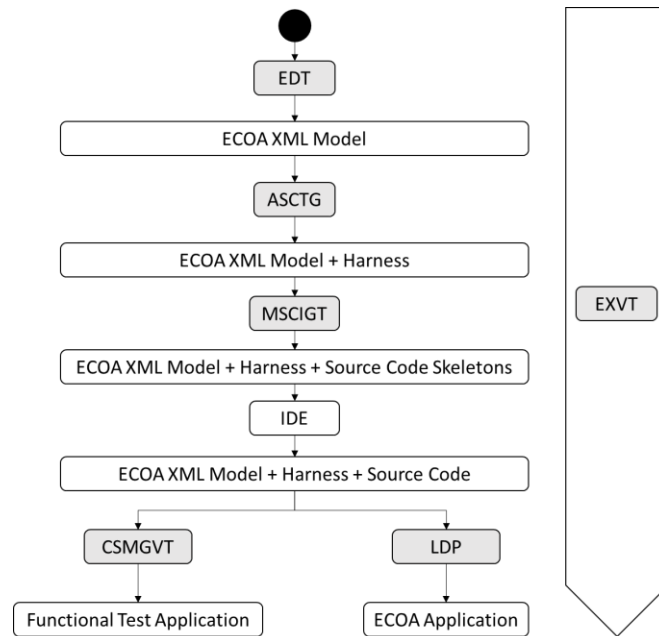


Figure 12: Use of tools in the development process

Note that you have two options to launch tools:

- Direct call to each tool (see user guides),
- Or use of the provided light user panel (see GUI documentation).

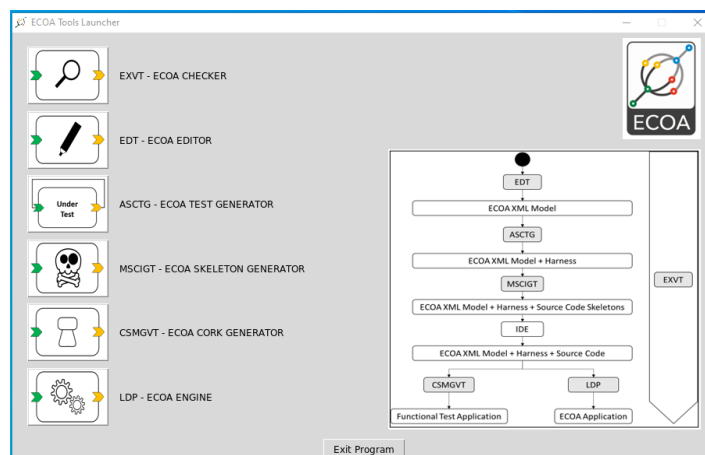


Figure 13: User Panel to launch ECOA tools

10.2.1 UC1: As a System Architect, I want to define the applicative architecture of my system

I have to launch **EDT** tool and create a new project. I use the tree view to create types, services and components definition (right click on an element to get a contextual edition menu).

For each element, all attributes to be defined are presented in the properties window, where they can be edited.

For the initial assembly schema, I can either work in the tree view or in the graphical view. I can instantiate already defined components, and connect provided services to required services.

To get a correct system, I need to know ECOA concepts, but I do not need to master the XML syntax of ECOA models.

At any time, I can choose to save my project and ask for an XML export according to ECOA AS6 standard.

A set of light consistency controls are automatically done on the fly during edition, but more complete tests are executed at export time by an implicit call to **EXVT** tool.

In case of errors, a detailed report is produced in the user interface to help correcting the system definition.

Note that if a change is done in a definition, it will be automatically propagated to all dependent objects, except for canceling operations that require a manual but guided procedure.



PRACTICE : Creation of types, services and components definition for “Rescue Device Driver” and “Rescue Tasks Execution” with **EDT**

Execution steps :

- Launch EDT
- Create a new project

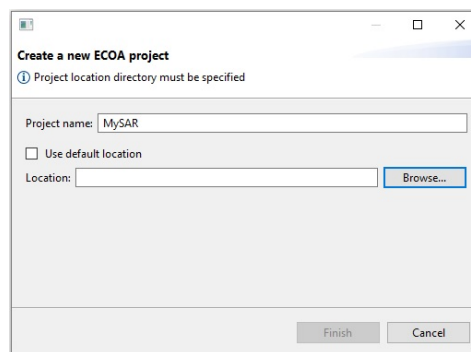


Figure 14: EDT New project creation (Tutorial UC1)

- Create types then services then components definitions in the tree view (right click on an object to create/edit sub-objects then in the Properties window to access to an object attributes).

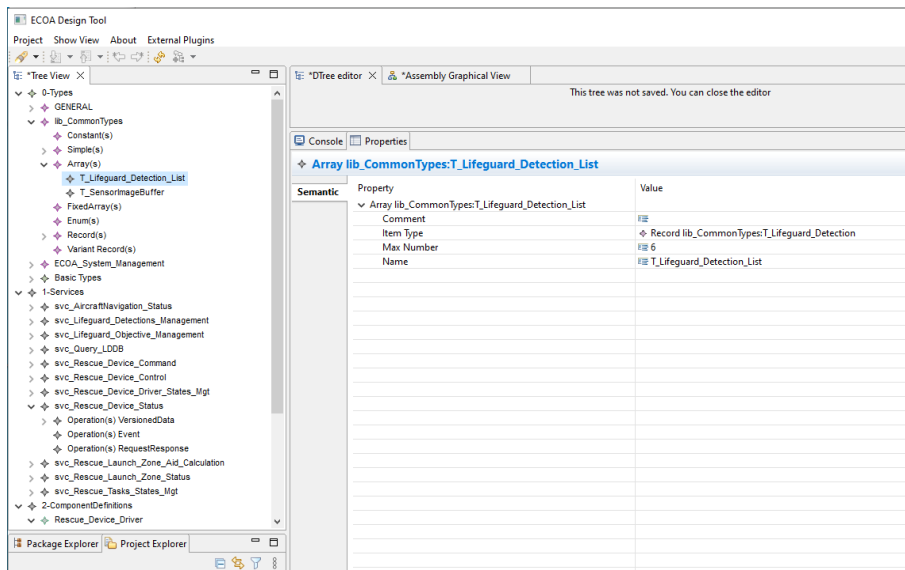


Figure 15: EDT Tree View of SAR example (Tutorial UC1)

- Save the project and select XML export

Results:

- EXVT is executed and the XML analysis report is displayed in the console window. If there is no errors, an ECOA XML model is delivered (some directories might be empty in accordance with the level of elements defined in EDT)

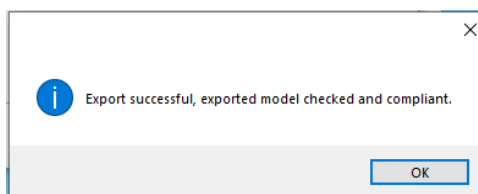


Figure 16: EDT screenshot of a successful export (Tutorial UC1)

Nom	Modifié le	Type	Taille
0-Types	11/04/2023 14:04	Dossier de fichiers	
1-Services	11/04/2023 14:04	Dossier de fichiers	
2-ComponentDefinitions	11/04/2023 14:04	Dossier de fichiers	
3-InitialAssembly	11/04/2023 14:04	Dossier de fichiers	
4-ComponentImplementations	11/04/2023 14:04	Dossier de fichiers	
5-Integration	11/04/2023 14:04	Dossier de fichiers	
SearchAndRescue.project.xml	11/04/2023 14:04	Document XML	2 Ko

Figure 17: EDT export of the ECOA model (Tutorial UC1)

- In case of errors, the EXVT report is displayed and gives information to locate them, so that the user might update the EDT project and try a new XML export.

10.2.2 UC2: As a System Architect, I want to modify an existing applicative architecture

In this use case, I am given an ECOA input model that needs to be updated. I may be not familiar with ECOA XML files, or just willing to easily make required changes. I just have to launch **EDT** tool, create a project and import the ECOA model. **EXVT** can be chosen as the required model checker for imported files: in case the input model is not complete, I have to specify the definition level to be covered by consistency tests.

If there is no error, the input model appears in **EDT** user interface: in the tree view, and, if an assembly schema is defined, in the graphical view.

EDT behavior is then the same that for the previous use case.



PRACTICE : Addition of an assembly schema for “Rescue Device Driver” and “Rescue Tasks Execution” using **EDT**

Execution steps :

- Launch EDT
- Open the project saved in UC1
- EXVT is executed and the XML analysis report is displayed in the console window : 0 error / 0 warning
- ECOA model elements are displayed in the tree view
- In the assembly graphical view, instantiate a component “Rescue_Device_Driver” from associated component definition, and a component “Rescue_Tasks_Execution” from the associated component definition
- In the graphical view, create links between the two component instances services.
- You can adjust the objects layout on the graphical view.

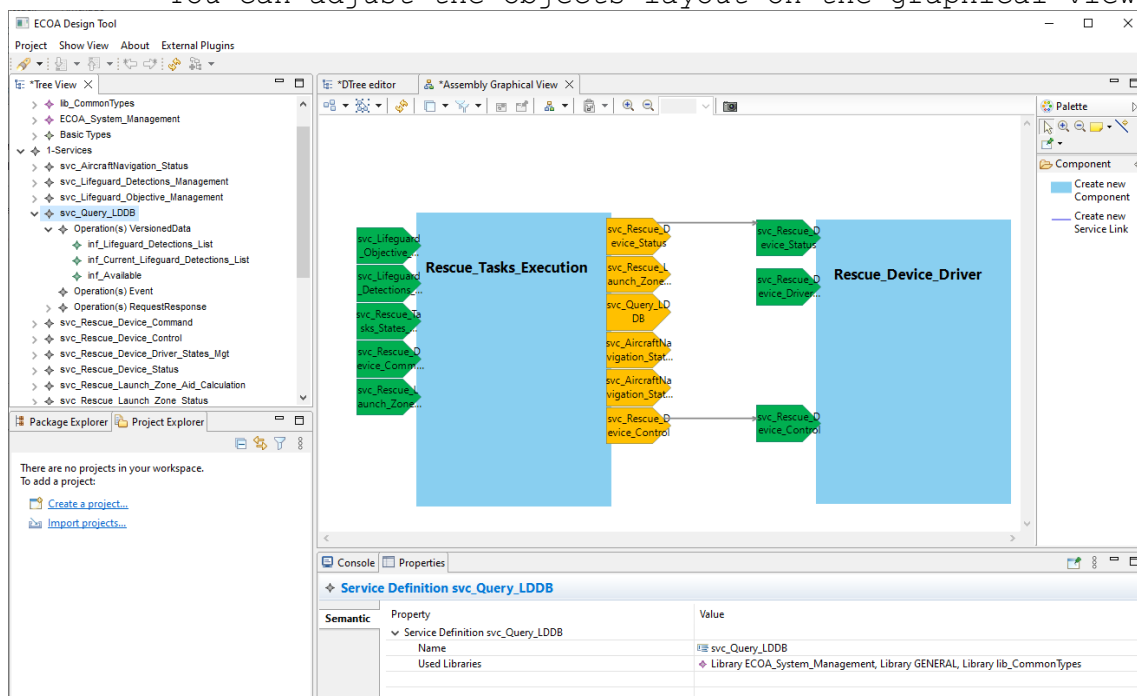


Figure 18: EDT screenshot of SAR example (Tutorial UC2)

- Save the project (including objects layout)
- Select XML export

Results:

- EXVT is executed and the XML analysis report is displayed in the console window.
- If there is no errors, an ECOA XML model is delivered.

10.2.3 UC3: As a System Integrator, I want to define components implementation architecture

I have to launch EDT, and import an ECOA input model that defines the applicative architecture containing my components. Components implementation can be edited either in the tree view or in the graphical view.

Note: the tool's purpose is only to capture components breakdown into modules in an ECOA model. It will not to assist the user in the design process defining this breakdown.



PRACTICE : Creation of "Rescue Tasks Execution" internal architecture in EDT

Execution steps :

- Launch EDT
- Open the project saved in UC2
- EXVT is executed and the XML analysis report is displayed in the console window : 0 error / 0 warning
- ECOA model elements are displayed in the tree view and in the assembly graphical view
- Create a component implementation for Rescue Tasks Execution component in the tree view, and create a module type, then a module implementation then a module instance and a trigger instance

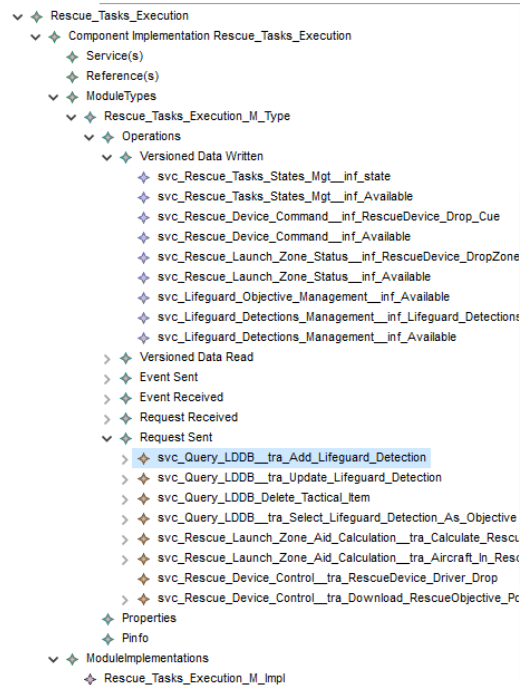


Figure 19: EDT - RTE implementation treewiew (Tutorial UC2)

- In the implementation graphical view for Rescue Tasks Execution component, instantiate the created module instance and connect module operations to corresponding

component services operations, then connect the trigger instance to the module Activate operation

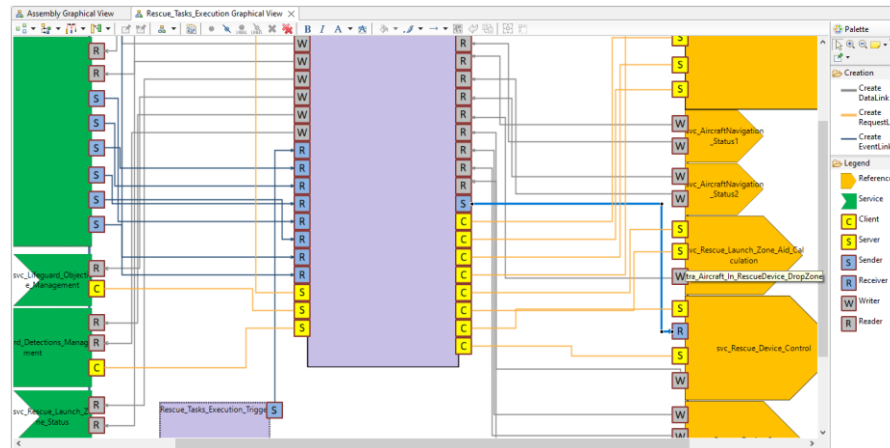


Figure 20: EDT screenshot of RTE implementation (Tutorial UC2)

- Save the project (including objects layout)
- Select XML export

Results:

- EXVT is executed and the XML analysis report is displayed in the console window.
- If there is no errors, an ECOA XML model is delivered.

10.2.4 UC4: As an ASC Supplier, I want to get an ECOA model specifically adapted to my components development

I am responsible for the development of a subset of components in the ECOA applicative architecture. I probably do not own components that are connected to mine. So, to be able to test my components, I need to define a specific new component (called “harness” component) replacing all these missing components.

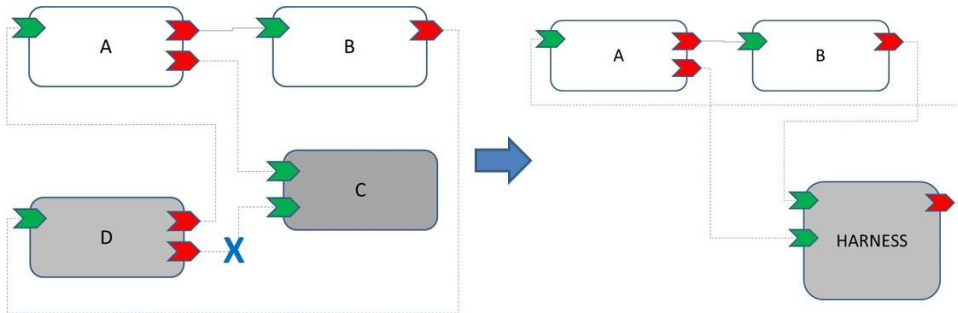


Figure 21: The harness component replaces components connected to a subsystem

From a configuration file listing the components subset to be tested, **ASCTG** allows to automatically adapt the input ECOA model by:

- Adding the definition and the implementation of the harness component,
- Defining a new assembly limited to my components and their harness component,
- Defining the associated new deployment (based on simplified hypotheses).

The new ECOA model I get offers me all the definition elements I need to complete my development.



PRACTICE : Creation of a Harness component to test “Rescue Device Driver” and “Rescue Tasks Execution” components, using **ASCTG**

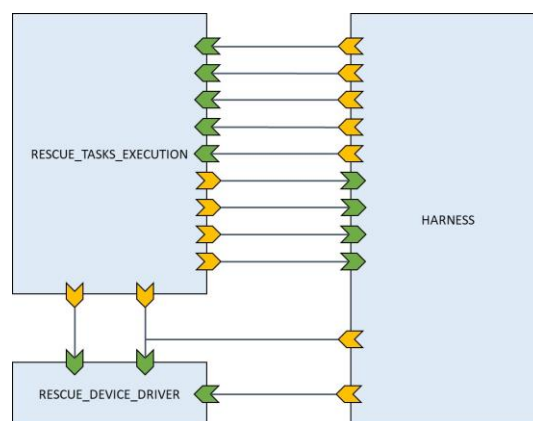


Figure 22: Expected harness to be generated by ASCTG (principle)

Execution steps :

- Prepare a configuration file for the subsystem under test

```

<asctg>
  <components>
    <componentInstance>Rescue_Tasks_Execution</componentInstance>
    <componentInstance>Rescue_Device_Driver</componentInstance>
  </components>
</asctg>
~
~

```

Figure 23: ASCTG configuration file (Tutorial UC4)

- Launch ASCTG with SAR example and the prepared configuration file

Results:

- EXVT is executed and reports 0 errors and 0 warnings
- A new component HARNESS is created with its XML implementation file

```

<?xml version='1.0' encoding='utf-8'?>
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0" componentDefinition="HARNESS">
  <use library="ECOA_System_Management"/>
  <use library="GENERAL"/>
  <use library="lib_CommonTypes"/>
  <moduleType name="MOD_HARNESS_mod_type">
    <operations>
      <eventReceived name="HARNESS_Trigger"/>
      <dataWritten name="Rescue_Tasks_Execution_svc_Query_LDDb_inf_Lifeguard_Detections_List" type="lib_CommonTypes:T_Lifeguard_Detection_List"/>
      <dataWritten name="Rescue_Tasks_Execution_svc_Query_LDDb_inf_Current_Lifeguard_Detections_List" type="lib_CommonTypes:T_Lifeguard_Detection_List"/>
      <dataWritten name="Rescue_Tasks_Execution_svc_Query_LDDb_inf_Available" type="ECOA:boolean8"/>
      <requestReceived name="Rescue_Tasks_Execution_svc_Query_LDDb_tra_Add_Lifeguard_Detection">
        <input name="Tactical_Item" type="lib_CommonTypes:T_Lifeguard_Detection"/>
        <output name="Tactical_Item_Added" type="ECOA:boolean8"/>
      </requestReceived>
      <requestReceived name="Rescue_Tasks_Execution_svc_Query_LDDb_tra_Update_Lifeguard_Detection">
        <input name="Tactical_Item" type="lib_CommonTypes:T_Lifeguard_Detection"/>
        <output name="Tactical_Item_Updated" type="ECOA:boolean8"/>
      </requestReceived>
    </operations>
  </moduleType>
</componentImplementation>

```

Figure 24: Extract of HARNESS implementation file (Tutorial UC4)

- New assembly for components Rescue Device Driver, Rescue Execution Task and HARNESS
- New deployment for these components

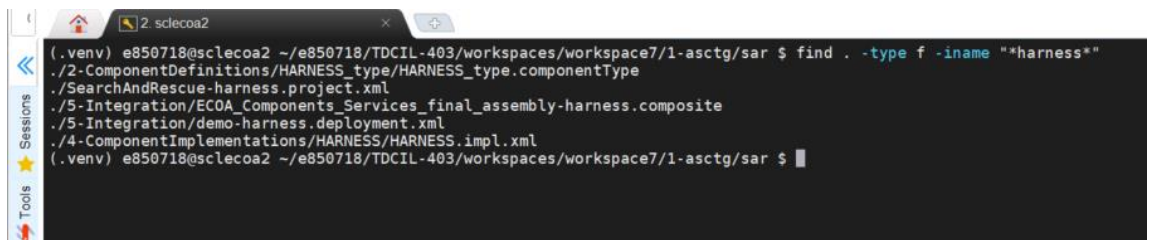


Figure 25: Generated files (Tutorial UC4)

Let us call SAR-HARNESS the updated ECOA model.

10.2.5 UC5: As an ASC Supplier, I want to generate source code for my component(s) modules

Once an ECOA model is defined from types to components implementation, I can launch **MSCIGT** to generate applicative source code, mainly corresponding to types declaration and modules entry-points functions.

A module entry-point is associated to a module input operation, and describes treatments to be executed for this operation. Depending on whether the activating parameter of the operation is true, the associated entry-point is called immediately or not when the operation is received.

Thanks to **MSCIGT**, several applicative source code files are automatically generated from the ECOA model:

- Defining the associated new deployment,
- Types declaration files,
- And for each module implementation:
 - Several declaration files addressing user data context, container functions and module entry-points functions,
 - An implementation file, which contains the so-called “code skeleton” because all module entry-points functions are defined with an empty body. I will have then to manually fill these functions bodies to complete the module behavior.

MSCIGT also generates elements for module unit testing composed of a unit test, a container mockup, and code building directives.

Container mockup code mainly contains empty functions (i.e. “skeletons”) to be manually filled, and allows then to test the module in a minimal environment.

Code generation is available for C and C++ languages.

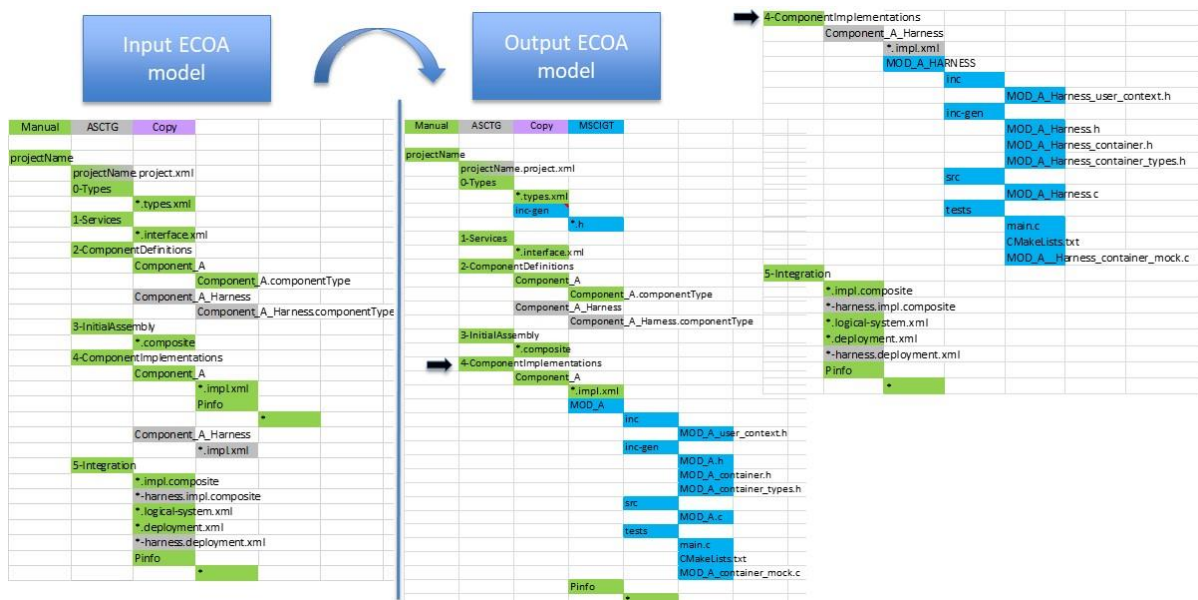


Figure 26: New files created after using ASCTG and MSCIGT



PRACTICE : Code skeletons generations for modules defined in “Rescue Device Driver”, “Rescue Tasks Execution” and associated “Harness” component with **MSCIGT**

Execution steps :

- Prepare a template file to define your file headers style
- Launch MSCIGT with SAR-HARNESS example and the prepared template file

Results:

- EXVT is executed and reports 0 errors and 0 warnings
- Source code files are created for all modules defined in components Rescue Device Driver, Rescue Execution Task and HARNESS. Particularly, module entry-points skeletons with comments to indicate where manual functional code must be inserted.

```
109
110 /**
111  * @internal    Insert function description here.
112  * @param[in]  context Insert variable description here.
113  * @param[in]  Power_Command Insert variable description here.
114  */
115 void
116 Rescue_Device_Driver_M_Impl__svc_Rescue_Device_Control__cmd_RescueDevice_Driver_Power_Control__received (
117     Rescue_Device_Driver_M_Impl__context * context,
118     const GENERAL__T_eCommanded_Power_State Power_Command
119 )
120 {
121     (void) context;
122     (void) Power_Command;
123     // Insert logic here.
124 }
```

Figure 27: Extract of RDD module source code (Tutorial UC5)

10.2.6 UC6 : As an ASC Supplier, I want to test one or several module(s) on the basis of my ECOA model and source code

- Case A: I only want to execute functional tests and/or I do not have an ECOA environment on my workstation.

It is possible to begin with unit tests for each of my modules: as we have seen in §10.2.5, **MSCIGT** generates source code files in that purpose, and I just have to fill in test functions bodies to implement my tests (and expected logs).

I can also use **CSMGVT** to execute functional tests by simulating a non real-time execution of several ECOA components. This is also interesting as a pre-integration test for my components.

- Step 1: As I probably do not own all the components that are connected with those I want to validate, I can use **ASCTG** to create the “harness” component that will be connected to the component(s) which contain(s) the module(s) under test. (cf. 10.2.4).
- Step 2: I generate source code skeletons for my harness component thanks to **MSCIGT**. (cf. 10.2.5)
- Step 3: In harness component implementation, I fill in harness module entry points with my source code, according to the behaviour I expect. For example, I can use an input file (PINFO) to define input data to be used in my validation scenario, and the associated expected results. I can execute one test each time the harness module is triggered, and log the comparison status between received results and expected ones.
- Step 4: I can use **CSMGVT** to execute components in my workstation environment, and check their functional behaviour.

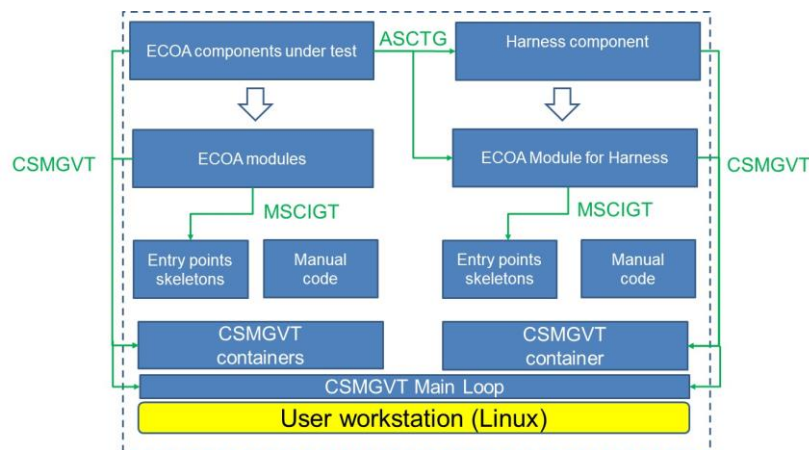


Figure 28: Creation of artefacts for functional tests with CSMGVT

PRACTICE : Execution of “Rescue Device Driver”, “Rescue Tasks Execution” and associated “Harness” component with **CSMGVT**

This document is prepared provided solely on an ‘as is’ basis and developers of this document make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.



Execution steps (1) :

- Insert manual code in artefacts generated by MSCIGT (not functionally significant in our SAR-HARNESS example)

```
void
Rescue_Tasks_Execution_M_Impl_container__svc_Rescue_Device_Control__cmd_RescueDevice_Driver_Power_Control__send (
    Rescue_Tasks_Execution_M_Impl_context * context,
    const GENERAL__T_eCommanded_Power_State Power_Command
)
{
    if (context->platform_hook->mod_id == RESCUE_TASKS_EXECUTION_AM_RESCUE_TASKS_EXECUTION_ID)
    {
        Rescue_Device_Driver_M_Impl__svc_Rescue_Device_Control__cmd_RescueDevice_Driver_Power_Control__received (
            &Rescue_Device_Driver_AM_Rescue_Device_Driver_Context,
            Power_Command
        );
    }
}
```

MANUAL SOURCE CODE

Figure 29: Extract of RTE module source code with manual insertion (Tutorial UC6)

- Launch CSMGVT with SAR-HARNESS example enriched with manual code
- Compile and execute

Results:

- EXVT is executed and reports 0 errors and 0 warnings
- Stubbing code files are created

```
— HARNESS
  └─ MOD_HARNESS_mod_impl
     └─ CMakeLists.txt
— Rescue_Device_Driver
  └─ Rescue_Device_Driver_M_Impl
     └─ CMakeLists.txt
— Rescue_Tasks_Execution
  └─ Rescue_Tasks_Execution_M_Impl
     └─ CMakeLists.txt
— src
  └─ CSM_SearchAndRescue-harness.cpp
     └─ main.cpp
— System_Manager
  └─ System_Manager_M_Impl
     └─ CMakeLists.txt
```

Figure 30: List of files created by CSMGVT (Tutorial UC6)

```

34 int main(void)
35 {
36     cm_initialize();
37
38     /* Initializing the ECOA modules. */
39     Rescue_Device_Driver_M_Impl__INITIALIZE__received(&Rescue_Device_Driver_AM_Rescue_Device_Driver_Context);
40     Rescue_Tasks_Execution_M_Impl__INITIALIZE__received(&Rescue_Tasks_Execution_AM_Rescue_Tasks_Execution_Context);
41     MOD_HARNESS_mod_impl__INITIALIZE__received(&MOD_HARNESS_mod_inst_HARNESS_Context);
42
43     // Insert arguments logic here.
44

```

Figure 31: Extract of main code generated by CSMGVT (Tutorial UC6)

Execution steps (2) :

- Launch SAR-HARNESS example compilation
- Launch SAR-HARNESS example execution

Results:

- Execution logs appear on your terminal

```

(.venv) e850718@sclecoa2 ~/e850718/TDCIL-403/workspaces/workspace7/4-csmgvt/sar-harness-csm/build $ ./csm
[INFO] RDD: Rescue Device (DAV)
[DEBUG] RDD: INITIALIZE
[DEBUG] RDD: New Component State STOPPED
[INFO] RTE: RTE (DAV)
[DEBUG] RTE: INITIALIZE
[DEBUG] RTE: New Component State STOPPED
[INFO] RTE_HARNESS: RTE_HARNESS (DAV)
[DEBUG] RTE_HARNESS: INITIALIZE
[DEBUG] RTE_HARNESS: New Component State STOPPED
[INFO] RDD_HARNESS: Rescue Device (DAV)
[DEBUG] RDD_HARNESS: INITIALIZE
[DEBUG] RDD_HARNESS: New Component State STOPPED
[DEBUG] RDD: START
[DEBUG] RDD: New Component State RUNNING
[DEBUG] RTE: START
[DEBUG] RTE: New Component State RUNNING
[DEBUG] RTE_HARNESS: START
[DEBUG] RTE_HARNESS: New Component State RUNNING
[DEBUG] RDD_HARNESS: START
[DEBUG] RDD_HARNESS: New Component State RUNNING

```

Figure 32: Example of CSMGVT lifecycle logs (Tutorial UC6)

```

[DEBUG] RDD: Component State RUNNING
[DEBUG] RTE: Component State RUNNING
[DEBUG] RTE: Weapon_Command.Weapon_Power_Control: Power_Command = OFF
[DEBUG] RDD: [Weapon_Preparation:RDD :Weapon_Control:cmd_RescueDevice_Driver_Power_Control:received
[DEBUG] RDD: [Weapon_Preparation:RDD :Weapon_Status:Weapon_Preparation_Status:write] WeaponStatus.Power_State = Off.
[DEBUG] RTE_HARNESS: svc_Query_LDDB_tra_Select_Lifeguard_Detection_As_Objective:Request received : 0
[DEBUG] RTE: svc_Query_LDDB_tra_Select_Lifeguard_Detection_As_Objective:Response received : 1
[DEBUG] RTE_HARNESS: Component State RUNNING
[DEBUG] RDD_HARNESS: Component State RUNNING

```

Figure 33: Example of CSMGVT functional logs (Tutorial UC6)

- Case B : I want to test my module(s) in an ECOA environment. This allows me to consider both functional and real-time issues.
 - Steps 1 to 3 are the same than in case A with **CSMGVT**, except that I can add real-time oriented tests.
 - Step 4: I can use **LDP** to execute both my component(s) and the harness component, according to logical-system and deployment files that were previously generated by **ASCTG**.

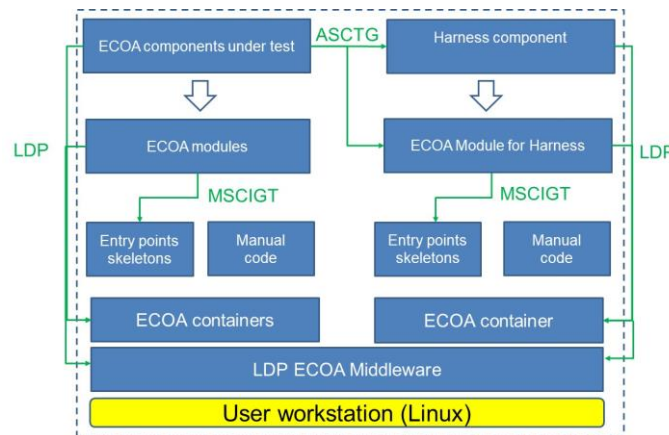


Figure 34: Creation of artefacts for tests in an ECOA environment



PRACTICE : Execution of “Rescue Device Driver”, “Rescue Tasks Execution” and associated “Harness” component with **LDP**

Execution steps (1) :

- Keep manual code from previous practicing exercise
- Launch LDP with SAR-HARNESS example enriched with this manual code

```

- Rescue_Tasks_Execution.impl.xml
- Rescue_Tasks_Execution_M_Impl
  - inc
    - Rescue_Tasks_Execution_M_Impl_user_context.h
  - inc-gen
    - DAV_Rescue_Tasks_Execution_M_Impl.h
    - Rescue_Tasks_Execution_M_Impl_container.h
    - Rescue_Tasks_Execution_M_Impl_container_types.h
    - Rescue_Tasks_Execution_M_Impl.h
  - src
    - Rescue_Tasks_Execution_M_Impl.c
  - src-gen
    - DAV_Rescue_Tasks_Execution_M_Impl.c
    - Rescue_Tasks_Execution_M_Impl_container.c
  - test
    - Rescue_Tasks_Execution_M_Impl_helpers.c
    - Rescue_Tasks_Execution_M_Impl_helpers.h
    - Rescue_Tasks_Execution_M_Impl_helpers.mak
    - Rescue_Tasks_Execution_M_Impl_helpers_test.c
    - Rescue_Tasks_Execution_M_Impl_test.c
    - Rescue_Tasks_Execution_M_Impl_test_container.c
    - Rescue_Tasks_Execution_M_Impl_test.mak
  - tests
    - CMakeLists.txt
    - main.c
    - Rescue_Tasks_Execution_M_Impl_container_mock.c

```

Figure 35: Some files generated by LDP (Tutorial UC6)


```

INFO      == Generate protection domains
INFO      # Generate C file for PD HARNESS_PD
INFO      # Generate C file for PD Rescue_Tasks_Execution_EXE
INFO      # Generate C file for PD Rescue_Device_Driver_EXE
INFO      ===== Completed

```

Figure 36: Generation of Protection Domains by LDP (Tutorial UC6)

- Compile and execute

Results:

- EXVT is executed and reports 0 errors and 0 warnings
- Execution logs are saved in dedicated files that you can read after execution.

```

["1679668596,831715777":1:"INFO":"main_PD":"main_node":"=== main started"
"1679668596,831794895":1:"INFO":"main_PD":"main_node":"thread_name: platform (27778) - sched_policy:0 - priority:0 - niceness:0 -
"1679668596,831934552":1:"INFO":"main_PD":"main_node":"=== Create process ./PD_HARNESS_PD (pid=27788)"
"1679668596,832041973":1:"INFO":"main_PD":"main_node":"=== Create process ./PD_Rescue_Tasks_Execution_EXE (pid=27789)"
"1679668596,832147895":1:"INFO":"main_PD":"main_node":"=== Create process ./PD_Rescue_Device_Driver_EXE (pid=27790)"
"1679668596,845810868":1:"INFO":"main_PD":"main_node":"[MAIN] accept new connection 30000"
"1679668596,846661298":1:"INFO":"main_PD":"main_node":"[MAIN] accept new connection 30002"
"1679668596,848757578":1:"INFO":"main_PD":"main_node":"[MAIN] accept new connection 30001"
"1679668597,846815699":1:"INFO":"main_PD":"main_node":"[MAIN] received client_init: 1"
"1679668597,847151695":1:"INFO":"main_PD":"main_node":"[MAIN] received client_init: 2"
"1679668597,847461043":1:"INFO":"main_PD":"main_node":"[MAIN] received client_init: 3"
"1679668597,847474491":1:"INFO":"main_PD":"main_node":"***** all components are connected"
"1679668597,847494460":1:"INFO":"main_PD":"main_node":"***Automatic Launch"
"1679668597,847700854":1:"INFO":"main_PD":"main_node":"[MAIN] received client_ready: 1"
"1679668597,847793557":1:"INFO":"main_PD":"main_node":"[MAIN] received client_ready: 2"
"1679668597,852828996":1:"INFO":"main_PD":"main_node":"[MAIN] received client_ready: 3"
"1679668597,852846402":1:"INFO":"main_PD":"main_node":"***** everybody is ready"

```

Figure 37: LDP execution dated logs about channels creation (Tutorial UC6)

Important: **CSMGVT**, and **LDP** as well, enable validation tests at component level only. So, I have to be careful when I am associating component level interfaces to module level interfaces for my tests (the easiest case is the bijective association for mono-module components).

10.2.7 UC7: As an ASC Supplier, I want to use binary files to integrate and test components

LDP generates binary files for ECOA modules from my available source code.

I can also receive binary files corresponding to my industrial partners' modules.

So, in my validation process, I want to integrate components with each other, only considering binary modules deliveries.

This work is possible with **LDP**, provided that the ECOA required bin-desc XML file is available for each module.

For my modules, bin-desc files have to be written manually (mainly to describe modules memory needs).

10.2.8 UC8: As a System Integrator, I want to define the software deployment onto hardware resources

I am in charge of defining the logical system and the deployment files in the ECOA model.

The logical-system description has to be representative of the target execution platform high-level architecture (particularly: number and characteristics of computation nodes). The deployment file addresses the definition of ECOA protection domains (processes with specific properties such as execution priorities) and their mapping onto hardware resources. These definitions are the result of my technical expert analysis, but I can capture them thanks to **EDT**, and be sure then to generate correct XML files, in accordance with ECOA AS6 standard.



PRACTICE : Deployment of modules defined in “Rescue_Device_Driver”, “Rescue_Tasks_Execution” and associated “Harness” component using **EDT**

Execution steps :

- Launch EDT
- Open the project saved in UC3
- EXVT is executed and the XML analysis report is displayed in the console window : 0 error / 0 warning
- ECOA model elements are displayed in the tree view
- In the tree view, create your logical-system
- Create your final assembly in the graphical view (instanciation of component definitions specifying the chosen component implementation)
- In the tree view, create a deployment for your modules by defining protection domains with their target node, and attached module/trigger instances
- Select XML export

Results:

- EXVT is executed and the XML analysis report is displayed in the console window.
- If there is no errors, an ECOA XML model is delivered.

10.2.9 UC9: As a Platform Provider, I want to provide an ECOA-compliant platform

I have to clearly identify the ECOA standard perimeter I want to cover.

In any case, I can get inspired by ECOA tools to create my ECOA adaptation layer.

If my platform already offers ECOA-like services, it may be sufficient to create an ECOA container generator that will call original platform services. **MSCIGT** and **CSMGVT** source code can give me good means to start this work.

If my purpose is more ambitious, I can ratherly analyze **LDP** implementation to get relevant ideas.

10.3 Highlights on ECOA standard benefits

Thanks to its formal approach based on an XML model, ECOA allows to create lots of tools which significantly enhance the applications development process:

- It avoids misunderstanding between partners as interfaces and associated mechanisms are non ambiguous,

- It increases productivity by generating technical source code (so that developers may focus on applicative-level issues).
As an example, we have experimented the development of an event-like exchange between two components in C language, reproducing the event mechanism apart from any ECOA environment. It gave the following results (not including header files):
 - ~250 lines of (reusable) middleware-level source code,
 - ~25 lines of send/receive code to be repeated for each event-like exchange, with risks of mistakes.
 - Loss of clearness in source code files because of the mix of technical and functional code (with no warranty on source code portability on future execution platforms).

- It increases validation efficiency by allowing the automatic creation of test harnesses and tests environments, targeting user different needs (unit, functional or real-time testing).

- It facilitates contractual exchanges of components between partners, based on well-defined binary and XML files.

This list can still be enriched as there are many other possibilities to be explored!

11 FAQ

- **How to begin with ECOA?**
A lot of examples are available on the ECOA website (section Documentation and Resources/Tutorials: see reference 2).
- **How can I get support to implement ECOA?**
It is planned to provide support to accompany the delivery of Open Source tools.
All related explanations will be available on the ECOA website: www.ecoa.technology