European Component Oriented Architecture (ECOA®) Collaboration Programme: Preliminary version of the ECOA Architecture Specification Part 3: Mechanisms

Dassault Ref No: DGT 2041082-A
Thales DMS Ref No: 69398917-035 --

Issue: 7

Prepared by
Dassault Aviation and Thales DMS

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

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Contents

Figures

0 Introduction

0.1 Executive Summary

The European Component Oriented Architecture (ECOA®) programme represents a concerted effort to reduce development and through-life-costs of the increasingly complex, software intensive systems within military platforms.

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

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

0.2 Main Introduction

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

This document is Part 3 of the Architecture Specification; it acts as an introduction to ECOA®.

IMPORTANT: The current version of the document focuses on **ECOA Core mechanisms**. Optional mechanisms are described in Architecture Specification Optional Parts.

The document is structured as follows:

- Section 6 provides an overview of ECOA® mechanisms.
- Section 7 describes in details mechanisms involved in the cope of an ECOA® Software Platform.
    - Section 7.1 describes the Component Lifecycle (i.e. runtime state management)
    - Section 7.2 describes Component interactions within an Application
    - Section 7.3 describes the Component mechanisms.
    - Section 7.4 describes the Composite and its Assembly
    - Section 7.5 describes Application characteristics
    - Section 7.6 describes interactions between Applications
    - Section 7.7 describes Infrastructure services for Application management
    - Section 7.8 describes building and deploying mechanisms
    - Section 7.9 describes the support for scheduling within an ECOA® system.
    - Section 7.10 describes platform utilities

1 Scope

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

2 Warning

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

3 Normative References

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

4 Definitions

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

5 Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
FIFO	First In, First Out
GUI	Graphical User Interface
ID	Identifier
OS	Operating System
PINFO	Persistent Information
QoS	Quality of Service
UDP	User Datagram Protocol
XML	eXtensible Markup Language
XSD	XML Schema Definition

6 ECOA Mechanisms

The Architecture Specification Part 1 defines an architecture which uses Components to build an Application. This document describes the mechanisms defined by ECOA and the way that Components interact within an Application. Additionally, it describes the behaviour of other aspects of an ECOA software architecture including management and utility functions.

In accordance with flexibility and openness principles, ECOA mechanisms are grouped in two categories:

- Core mechanisms that characterize ECOA technology, and that rely on a simple common set of concepts,
- Optional mechanisms that allow to fit ECOA to different user profiles (for example: ability to supervise the lifecycle of components).

The intended audience for this reference manual is:

- Component Developers:
  - To understand the mechanisms available for developing applications
- ECOA Platform Developers:
  - To understand the behaviour an ECOA Platform is required to provide for a given mechanism

This document describes the mechanisms available to a Component, but it is the Architecture Specification Part 4 which provides the abstract API for implementing the mechanisms described herein.

General Considerations

The behaviour of an ECOA platform consists of four main stages:

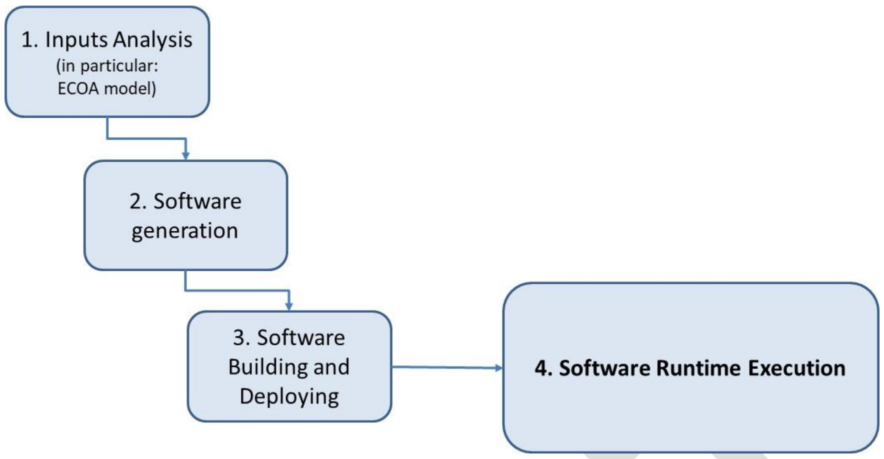


Figure 1 ECOA platform behaviour overview

This document mainly focuses on stage 4 dealing with runtime execution mechanisms. As there is a strong level of entanglement between concepts related to component lifecycle, component interactions and component internal mechanisms, it is recommended to have previously read Architecture Specification Parts 1 and 2 to have a good understanding of the following sections.

Architecture Specification Part 5 gives more details on platform requirements about other stages.

7 Scope of an ECOA Software Platform

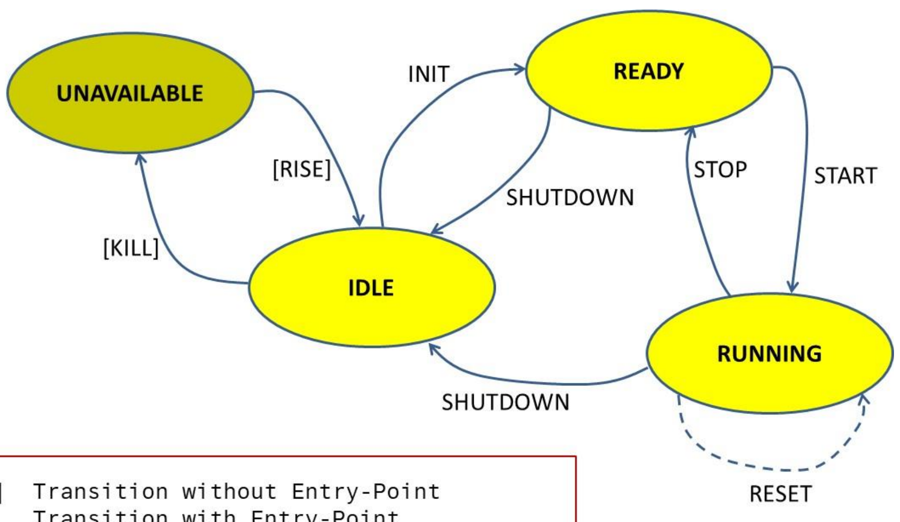
7.1 Component Lifecycle

Component run-time behaviour is dependent upon the Component Runtime Lifecycle state.

The Runtime Lifecycle of a Component Instance is a state-transition diagram which defines the logic of its runtime state changes according to a set of predefined Component Operations called Lifecycle Operations.

Note that all references to SUPERVISOR components in this section only apply to Infrastructures offering the [OPTION SUPERVISION].

Figure 2 illustrates the Component Runtime Lifecycle.



```
[x]  Transition without Entry-Point
 x   Transition with Entry-Point
(x)  Transition with optional Entry-Point
```

Figure 2 Component Runtime Lifecycle

A Component Instance has four possible logical states:

- UNAVAILABLE – default state for components not yet created (only known from ECOA Infrastructure or SUPERVISOR components)
- IDLE – state reached after component creation, or reached after shutdown.
- READY – state reached when the component has been initialized.
- RUNNING – state where the Component Instance is handling incoming Operations. In the RUNNING state, the Component Instance may be blocked when invoking Container Interface blocking operations such as synchronous request-responses.

A Component Instance can transition between these states as shown in Figure 2.

The UNAVAILABLE state is only known to Infrastructure and SUPERVISOR components; it is meaningless for the Component itself. KILL and RISE transitions are not part of Components Interface: they can only be triggered by Infrastructure and correspond to Executable management actions (launch/exit). After a RISE transition on a Component, the ECOA Infrastructure notifies implied SUPERVISOR components as soon as the newly created component is available (in IDLE state).

Other states and transitions of Component Lifecycle are managed by the Container, which invokes Component Interface entry points for each state change. The Component Instance states do not necessarily reflect the states of the underlying OS tasks which support the Component Instances, e.g. a Component Instance may be in a RUNNING state while the underlying task may be in a ready state waiting for the CPU resource.

The Component Interface always contains the four following entry points:

- INIT
- START
- STOP
- SHUTDOWN

These entry points are invoked by the Container as a result of a Lifecycle state change (such as the end of initialization), at the request of Infrastructure or SUPERVISOR components.

When a RUNNING Component raises a fatal error (see section 7.7.3), the ECOA Infrastructure applies a SHUTDOWN transition to this Component lifecycle.

An optional RESET entry point can be added in the Component Interface using the Option element `hasReset` in the ComponentImplementation.

The RESET transition does not change the Component Lifecycle state (only possible from RUNNING state to RUNNING state). It is useful to easily set the Component functional state back to a specific well-defined configuration (typically for testing purpose).

RESET transition is never triggered at the ECOA Infrastructure initiative. It is managed by SUPERVISOR components or, possibly, depending on Infrastructure extensions, by monitoring tools (example: Components control panel).

Please refer to section 7.7.1 to see how Components Lifecycle is managed by infrastructure during start-up, runtime and shutdown phases.

7.2 Interactions within an Application

Interactions between Components in an ECOA Application rely on three primary communication mechanisms:

- Events
- Request-Response
- Versioned Data

Thus, a Component Type defines Operations (so-called “normal” Operations) based on these mechanisms.

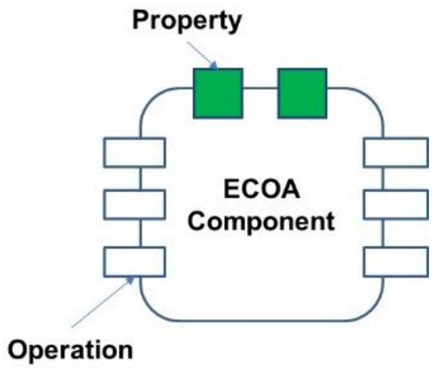


Figure 3 Generic view of ECOA Component Operations

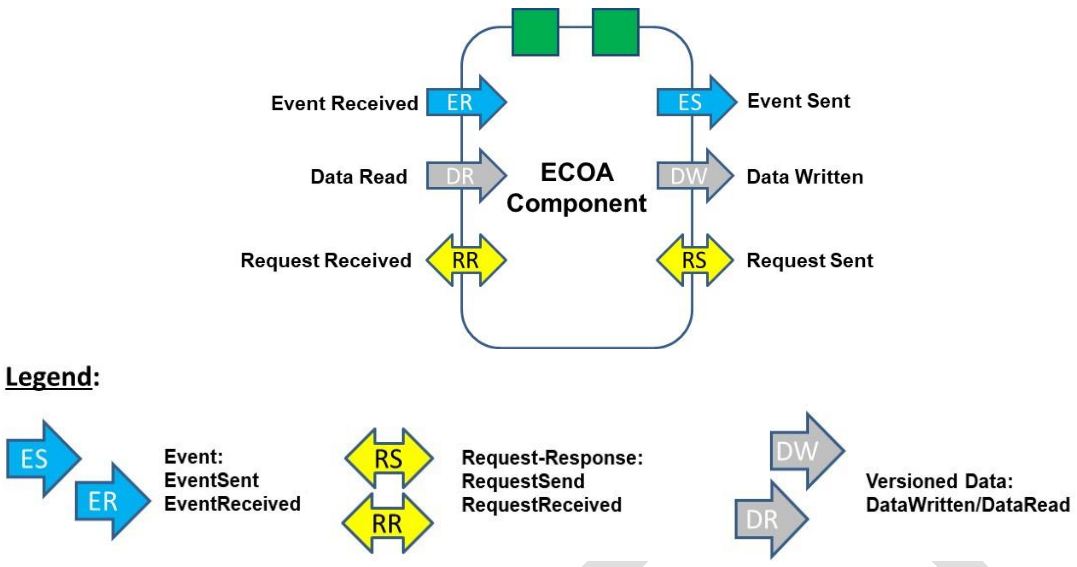


Figure 4 Graphic representation of normal Component Operations

Unlike Component Lifecycle Operations which are handled in any state (to enable the Lifecycle of a Component to be managed), normal Component Operations are only handled in the **RUNNING** state. Nevertheless, for a given Component Instance, Lifecycle Operations and normal Component Operations are placed in the same Component Instance Queue, which is managed according to rules detailed in section 7.3.1.

Component interactions can be defined between Components of different levels:

- Between Concrete Components,
- Between a Concrete Component and a Composite,
- Between Composites.

Any interaction defined for a Composite is associated to an interaction defined for a sub-component of this composite. For detail on the behaviours, see section 7.4.

Figure 5 gives a simple illustration of possible interactions between Components:

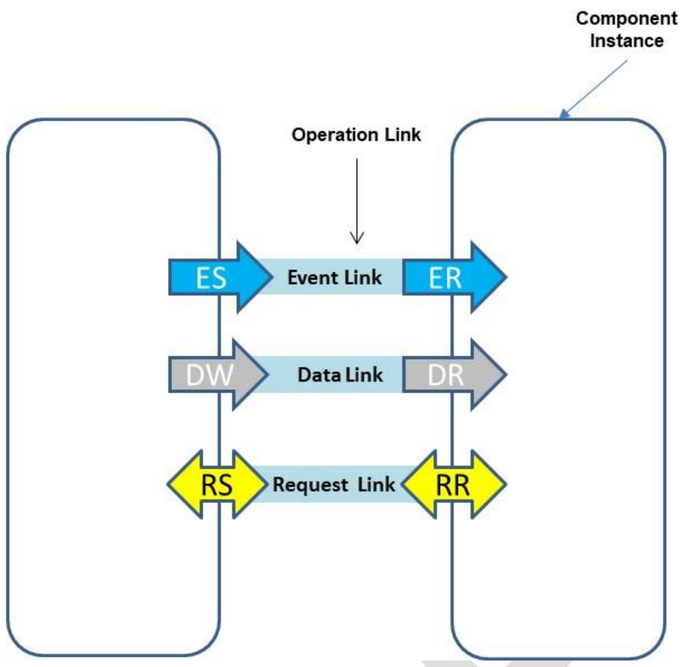


Figure 5 ECOA Interactions – Link level view

About previous figures:

- Arrows indicate data flow direction of Operations. In case of a Request-Response, two data flows of opposite directions are defined, corresponding to Request and Response exchanges.
- Container Operations can be connected to Component Operations using Operation Links. Note that some platforms may support optional Conditional Links (see Architecture Specification Optional Part related to [OPTION SUPERVISION]).
- All Component Operation Links require Container code.

The following sections include numerous figures, which illustrate the interactions within an ECOA software architecture and provide visual clarity. The key shown in Figure 5 and Figure 6 (hereafter) offers guidance on the colouring and symbology used throughout these sections.

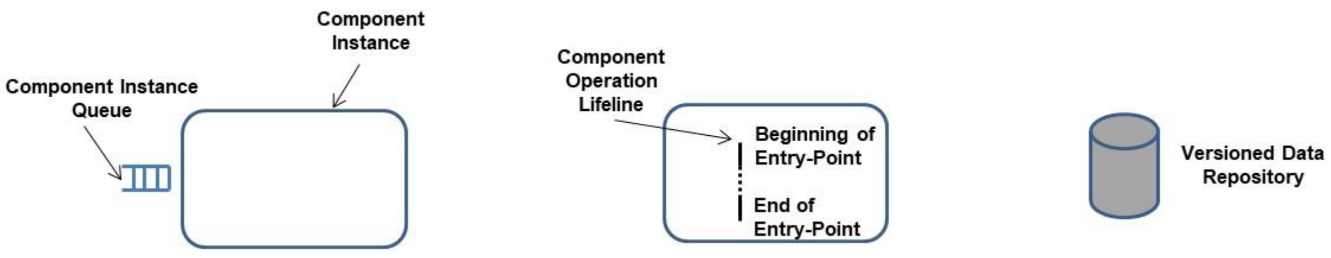


Figure 6 ECOA Interactions – Concepts representation

Explanations on concepts introduced in Figure 6 are given section 7.3.

Note that the Component developer is only responsible for implementing the functionality within the Component Instance. The other infrastructure objects shown are the responsibility of the ECOA Platform Developer and comprise the Platform Integration Code (e.g. Component Instance Queues, Versioned Data repositories, Operation Links etc.).

In addition to the above inter-components communication mechanisms, different API exist for Infrastructure Services to allow the management of properties, logging, local time service, persistent information, user context management and optional mechanisms when necessary.

7.2.1 Event

The Event mechanism is used for one-way asynchronous “push-style” communication between Component Instances and may optionally carry typed data.

For each Event, there is one or several Component Instances defined as Senders, and there may be multiple Component Instances referred as Receivers, in which case instances of the Event are broadcast to all receivers.

Events are “wait-free” and “one-way”: the Sender is never blocked and does not receive any feedback from Receivers. Events arriving on a full Receiver queue are lost, and the fault is reported to the fault-management Infrastructure.

If an Event arrives at a Component Instance that is not in the **RUNNING** state, the Event is discarded silently.

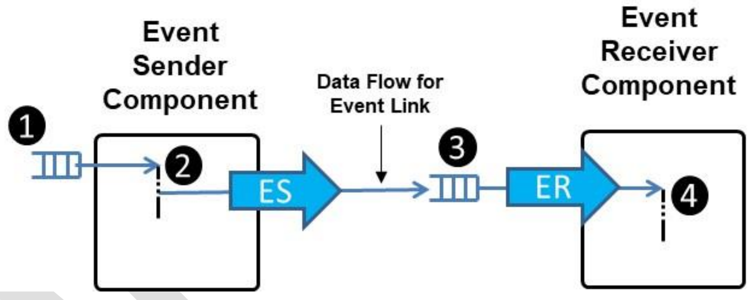


Figure 7 Event sent and received

Figure 7 shows, at **point 1**, an Operation being invoked on the Sender Component Instance as a result of some other activity. During this execution, the Component Instance performs an Event Send Container Operation at **point 2**. The Event Send operation returns immediately, allowing the initiating Component Instance to continue its execution. The Event will be queued on the Receiver Component Instance Queue shown at **point 3**. The Event Received Component Operation will then be invoked on the Receiver Component Instance when the queue is processed and the Event reaches the front of the queue, at **point 4**.

7.2.2 Request Response

The “Request-Response” mechanism is a two-way communication between Component Instances. The calling Component Instance requests an operation and the called Component Instance provides a Response.

The Requesting Component Instance (sender of the Request) is named the "Client", and the providing Component Instance (sender of the Response) is named the "Server".

A Request may carry data ("in" parameters) and the Response may also carry data ("out" parameters). All parameters are named and typed.

There are two mechanisms for Request operations which provide synchronous and asynchronous behaviour at the Client and one mechanism for Response operations at the Server. The details of these are described in the following sections.

For each Request-Response, the set of possible Clients and Server is identified at design time, as is whether the Client uses the synchronous or asynchronous mechanism. For a given Client, there is only one Server.

A Response from a Server is only sent to the particular Client that has issued the Request.

The client Container instance will implement a timeout (if defined at Component Type level) in order to unblock the Client of a Synchronous Request-Response or to inform the Client of an Asynchronous Request Response if no Response is received within the given timeout. The value of the timeout is defined at Component Type level. If the Response arrives after the timeout, the Response is discarded by the container and the fault is handled by the fault management. If the timeout is set to less than zero, it is considered as infinite; the Client of a Synchronous Request-Response remains blocked indefinitely if it never receives a Response.

A Request may fail if the Server is not available (e.g. it is not RUNNING, the remote platform is DOWN, network error). When such a failure can be detected by the Infrastructure, it returns with « no response » error code to the client, and reports the fault to the fault-management Infrastructure (if any). Where failures cannot be detected by the infrastructure, the client may use the timeout to avoid being blocked indefinitely.

7.2.2.1 Synchronous Request

In the case of a Synchronous Request, the Client Component Instance is blocked until the Response is received, as shown in Figure 8.

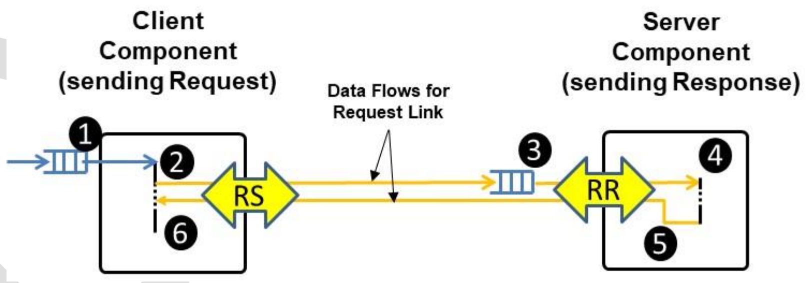

Client Component (sending Request)
Data Flows for Request Link
Server Component (sending Response)
1
2
RS
6
3
RR
4
5

Figure 8 Synchronous Client Request-Response

Figure 8 shows, at **point 1**, an Operation being invoked on the Client Component as a result of some other activity. During this execution, the Component Instance performs a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Component Instance becomes blocked. When blocked, the Client Component Instance does not handle any other incoming Component Operation. From a lifecycle point of view, this blocking situation is considered as a sub-state of the RUNNING state (see section 7.1).

The Request operation is connected via an Operation Link to the Server Component Instance, whereby the Request operation is queued in the Server Component Instance Queue, at **point 3**. The Request is accepted by the Server Component Instance as long as it has enough resources to handle the Response and it is in the **RUNNING** state. If not, the Request is discarded and a « no response » error code is returned to the client.

The Request operation will be invoked on the Server Component Instance at **point 4**, which, in this example, will send the Response at **point 5** just before completing the Request operation. Once the Response is received by the Client Component Instance, it will become unblocked and can continue its execution at **point 6**.

7.2.2.2 Asynchronous Request

In the case of an Asynchronous Request, the Client is released as soon as the Request has been sent and may continue to execute other functionality. The Response results in the call of an operation on the Requesting Component Instance, as shown in Figure 9.

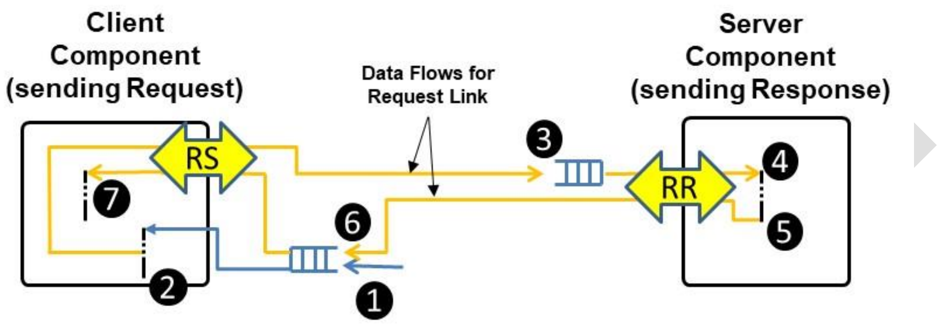

Client Component (sending Request)
Data Flows for Request Link
Server Component (sending Response)
RS
RR
1
2
3
4
5
6
7

Figure 9 Asynchronous Client Request-Response

Figure 9 shows, at **point 1**, an Operation being invoked on the Client Component Instance as a result of some other activity. During this execution, the Component Instance performs an Asynchronous Request operation at **point 2**. At the point the Request is made, the Client Component Instance does **NOT** block meaning it can finish its execution of the invoked operation.

The Request operation is connected via an Operation Link to the Server Component Instance, whereby the Request operation is queued in the Server Component Instance Queue, at **point 3**. The Request is accepted by the Server Component Instance as long as it has enough resources to handle the Response and it is in the **RUNNING** state. If not, the Request is discarded and a « no response » error code is returned to the client.

The Request operation will be invoked on the Server Component Instance at **point 4**, which, in this example, will send the Response at **point 5** just before completing the Request operation. The Response will then be placed in the Client Component Instance Queue at **point 6** as long as it is in the **RUNNING** state, and the Response call-back operation will be invoked on the Client Component Instance at **point 7**.

7.2.2.3 Deferred Response

By default, the Server decides when it replies to the Client. It can be done in the same block of functionality associated to the Request (see 7.2.2.1 and 7.2.2.2) or the Server may defer the provision of the Response e.g. where it needs to invoke a Request-Response Operation in order to provide the Response. In the second case the Server may continue to execute other functionality before providing the Response, as shown in Figure 10.

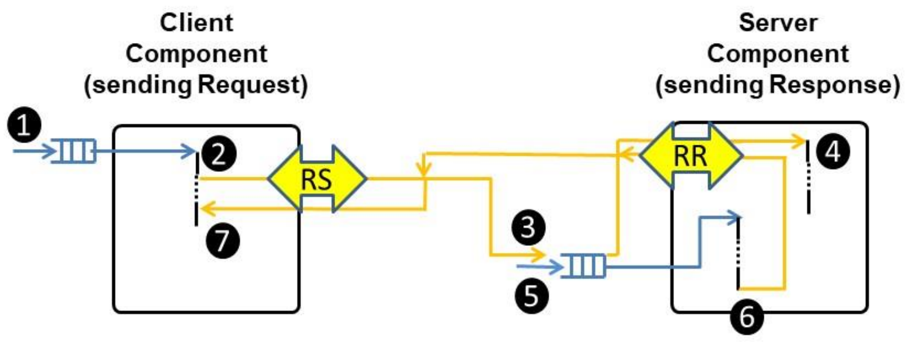


Figure 10 Deferred Response Server Request-Response

Figure 10 shows, at **point 1**, an Operation being invoked on the Client Component Instance as a result of some other activity. During this execution, the Component Instance performs, in this example, a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Component Instance becomes blocked.

The Request operation is connected via an Operation Link to the Server Component Instance, wherein the Request operation is queued in the Server Component Instance Queue, at **point 3**. The Request is accepted by the Server Component Instance as long as it has enough resources to handle the Response. If not, the Request is discarded.

The Request operation will be invoked on the Server Component Instance at **point 4**. The Server may, by design, use the Response Container Operation to send the response at some later time. For example, in order to compute the Response, it may be necessary to invoke a further Asynchronous Request operation, meaning the original Response cannot be computed until receipt of this Response occurs.

Point 5, shows another Operation invoked on the Server Component Instance, during this execution, at **point 6**, the Response Container Operation is called to send the Response back to the Client. Once the Response is received by the Client Component Instance at **point 7**, it will become unblocked and can continue its execution.

7.2.2.4 Immediate Response

When an immediate Response is required from the Server for a specific Request, the Server Component Instance automatically sends the Response to the Client as soon as treatments invoked by the Request Operation are completed (i.e. the Component Developer has no software interface to call for Response sent but behaviour is similar to an explicit sent of a deferred response at the end of the invoked Request Operation).

Immediate Response mechanism is available for both synchronous and asynchronous requests.

See Figure 8 and Figure 9 to illustrate those two cases.

7.2.3 Versioned Data

The Versioned Data mechanism allows Component Instances in an Application to share typed data. A reading Component Instance is named a “Reader”, and a writing Component Instance is named the “Writer”.

The data can be read by zero, one or more components.

The data can be written by zero, one or more components.

The Version Data allows access, at the Component's initiative, to the "current value" (or latest value) of a shared information. This is in contrast with the Event mechanism, which is more appropriate when the whole sequence of successive values (a stream of values) is of interest to the consumer of the information.

The Versioned Data mechanism is designed to offer the following important properties:

- **consistency**: a Component shall not be allowed to access a data value that is being modified by another component. In other words, every update of the data is *atomic* from the Component's point of view. The data is made available to readers entirely or not at all.
- **stability**: a Component shall keep access to a given value for as long as it wishes, without being impacted by any concurrent modifications of the data.
- **wait-free** (or non-blocking): a Component shall never be blocked when accessing a data.

The stability property implies a concept of delimited "access". Writers and Readers have to specify the beginning and the end of each (read or write) access to the data. A Component accesses a Version Data, as reader or as writer, in three steps:

1. **request for access** (via the Container Interface) and obtain a handle (i.e. logical "pointer") on a data value; this shall be non-blocking.
2. **use the data value**, using the handle.
3. **release the handle**, thus ending the access; this shall be non-blocking; in the case of a write access, there are two possibilities: either *publish* the changes made to the data value, either *cancel* them. After the handle has been relased, the data value shall not be used anymore by the Component, until another access is requested.

The properties of the Versioned Data mechanism imply that the Infrastructure usually has to manage multiple memory areas, each containing a certain "version" of the data, hence the name of the mechanism. The Infrastructure also has to make some copies of the data, at dedicated times, between these memory areas. The Components using the mechanism do not have to copy the data themselves. There are multiple possible implementations of the Versioned Data mechanism. The ECOA Architecture Specification specifies the semantics of the mechanism, but does not mandate or describe any specific implementation.

The term "*repository*" is used to designate the set of all versions (or copies) of a Versioned Data, managed by the Infrastructure, as well as the associated information needed to track the use of the different versions.

The Versioned Data mechanism, like other mechanisms, is defined at assembly level, and works with the same semantics in every deployment. For example, the properties of the mechanism described in this specification are applicable identically when Components are deployed in the same Executable and when they are deployed in different Executables.

7.2.3.1 Versioned Data read access

A Reader gets access to a memory area containing the latest data value at the time of the read request.

During the read access, this memory area is not affected by any subsequent changes to the data-set (i.e. by a Writer updating the data).

A read access ends when it is "*released*".

A Reader is not allowed to modify the data it has access to.

A read access request may fail if the data has no value because it never has been written.

7.2.3.2 Versioned Data write access

A Writer gets access to a memory area containing:

- either the latest data value at the time of the read request ("Read+Write" mode),
- either uninitialized memory ("Write only" mode).

The mode is selected according to a Metamodel attribute of the Component ("writeOnly", of boolean type). The default mode is "Read+Write".

In "Read+Write" mode, every writer is also a reader, which means the Component can make read accesses as well as write accesses.

During the write access, this memory area is not affected by any subsequent changes to the data-set (i.e. by another Writer updating the data).

NOTE: "Write Only" mode makes it explicit that a component cannot modify a data, which brings safety and explicitness to the software architecture. In addition, it avoids copying the latest data value for Writers which do not need to read it, thereby improving performance.

In a write access, local changes will not cause the Versioned Data to be updated until a "publish" is performed.

A write access ends with two possible alternatives:

- "*publish*" – modifications made by the Writer are published, i.e. become available for subsequent accesses.
- "*cancel*" – modifications made by the Writer are discarded, i.e. will never be visible by any access to the Versioned Data.

Data publications are atomic; the Infrastructure guarantees that "simultaneous" publications of the same data cannot corrupt the data content. All publications are serialized with a strict ordering, and the last publication determines the current value of the Versioned Data.

7.2.3.3 Versioned Data operation links

Read and Write operations are connected in an Assembly Schema through data links between Readers and Writers.

A single data link may have several readers and writers. All readers see all updates made to the data, by any writer.

It is also possible to connect a Write operation to several data links. In this case, a write will affect all data links.

If a Write operation is involved in several data links, and if this operation is in “Read+Write” mode, the assembly shall specify which link will be the "reference" one. This is necessary to determine which value is the one to be used for:

- the initialization of the memory area with the latest value, in Write accesses,
- Read accesses.

If a Write operation is involved in several data links, one and only one of the Write elements concerning this operation in these links shall have the "reference" attribute set to "true". Otherwise, an error shall be raised during the generation process.

It is not possible to connect a Read operation to several data links.

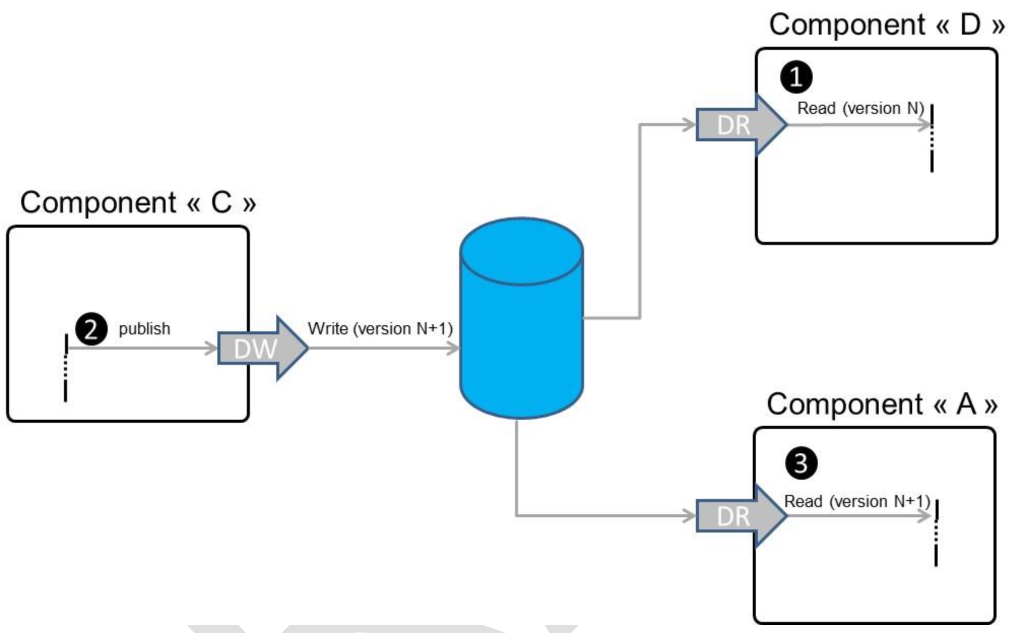


Figure 11 Versioned Data within an Application

Figure 11 shows, at **point 1**, Component Instance Reader “D” getting access to the Versioned Data and reading the latest value (N). At **point 2**, the Writer publishes a new version of the Versioned Data, that is then available at **point 3**, when the Component Instance Reader “A” requires access to the Versioned Data. At point 2 and afterwards, if component "D" still has the same access it got from point 1, it continues to see the same value; only the accesses started after point 2 are affected by the new publication.

7.2.3.4 “maxVersions” attribute

An optional attribute (maxVersions) may be set in the Component Type as an attribute of the data read/write operation to specify the maximum number of overlapping read or write accesses that may be performed by this Component. The number of read or write accesses is incremented each time the ECOA Infrastructure grants access to a Component Instance, and is decremented each time a Component Instance releases a read access or publishes/cancels a write access.

The default is one access per operation e.g. a Read must be released before the next Read starts, and a Write must be published or cancelled before the next Write starts. This is the most frequent use case, but other use cases are possible, where an access to a data is retained for a long time, and other accesses are started later, to other versions of the same data. This may be useful, for example, to analyse the evolutions of a data in time.

The number of overlapping read or write accesses must not exceed the **maxVersions** attribute value, otherwise the access request may fail. A read or write if a new local copy of the data cannot be created. In this case, a null Data Handle is returned, the Component is notified of the failure of the call, and the fault reported to the fault-management Infrastructure, if any. The maxVersion attribute allows the infrastructure to reserve enough memory to handle all possible access scenarios without failure, provided that each component behaves as declared.

In the following example, a Component accesses data D twice, which is legal if maxVersions is at least 2:

```
h1 = D_get_read_access()  // h1 = the current value of the data
// later:
h2 = D_get_read_access()  // h2 = the current value, may have changed since h1
// do some things with h1 and h2
D_release(h1)
D_release(h2)
```

7.2.3.5 Versioned Data stamp

The ECOA Infrastructure provides reader Component Instances with a stamp associated with the Versioned Data access. This stamp is available within the Versioned Data handle, for pure reader accesses, and also for write accesses that are not qualified as "write only". It allows a given Component to know if the data has been published (since an arbitrary previous access) or not. If the stamp's value is equal to a previous stamp value, the Component knows that the data has not been republished, and therefore that its value has not changed. Conversely, if the stamp values are different, a new value has been published, which means that the value of the data may have changed, but may also be identical. The ECOA Infrastructure hosting a reader Component Instance is responsible for updating the stamp of the Versioned Data.

The behavior of the versioned data stamp is the following and it is up to the platform supplier to choose how to manage the stamp value accordingly:

- The stamp value is “0” when the return status of a read operation is not OK.
- For any given reader, each time the data value is updated by a publish action by a writer, the stamp value shall be increased.
- The stamp may wrap around when reaching the max value of 2^{32}.

The stamp is a mechanism that is purely local to a read or write access. It implies that, for a given versioned data, different Component Instances may get different stamp values. Thereby, stamp values should not be compared between different readers, nor communicated between different Component Instances.

7.2.3.6 “uncontrolledAccess” attribute

For each Operation Link that uses this type of interaction, the Versioned Data mechanism can be configured to operate with or without access control.

With access control (the default behaviour), the ECOA Infrastructure ensures consistency and stability, as explained above.

Without access control, the ECOA Infrastructure does not ensure consistency and stability. Therefore, concurrency must be managed at application level by the Application Supplier.

Versioned Data without access control may be used to improve the performance of Applications in which large datasets are shared between several Component Instances, provided that concurrency is managed by other means.

The choice of enabling or disabling access control is made in the Assembly. It does not impact Components, which use the same models and API in both cases.

In the case of Versioned Data without access control, the ECOA Infrastructure maintains only one instance of the data in memory. This instance is directly shared between all components that require an access to the data. The infrastructure never makes any copies of the data when a Reader or a Writer accesses it. All components read and write at the same address.

Therefore, it is the Application responsibility to manage/synchronise/coordinate accesses to the Versioned Data. If no special care is taken, there is a possibility that a read and a write access happening concurrently result in an inconsistent memory state being seen by a component. In practice, it means that all Components accessing the data shall be coordinated in some way, statically by design, or dynamically by exchanges and synchronisations between components. For example, when all components are deployed on the same task,

7.2.3.7 Notifying Versioned Data

Versioned Data Readers can also specify an optional attribute (notifying) to receive a notification of any updates to Versioned Data. This behaviour is achieved by queuing a notification Event on the Reader Component Instance. This behaviour applies to Versioned Data with or without access control.

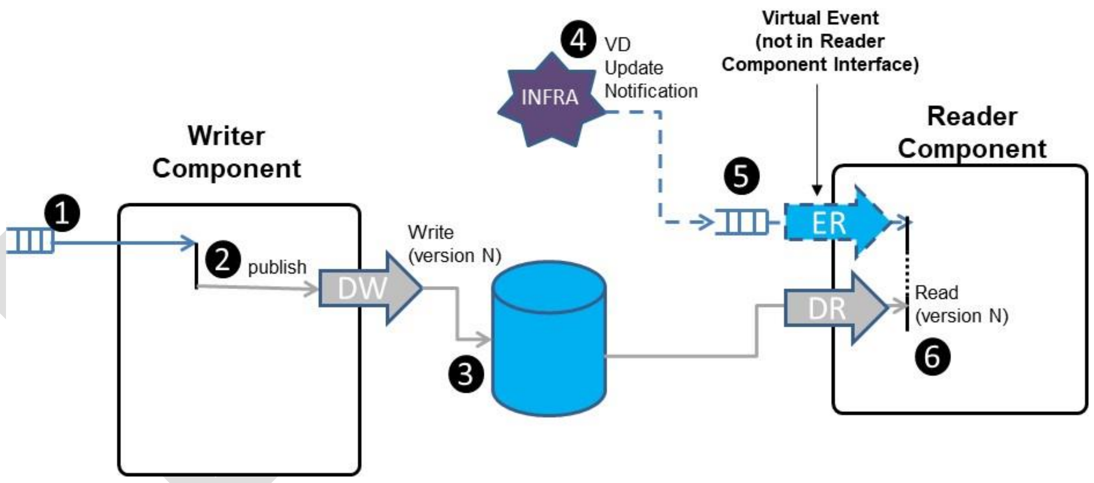


Figure 12 Notifying Versioned Data Behaviour – illustrated with access control

Figure 12 shows an example of Notifying Versioned Data with access control. At **point 1,** an Operation is invoked on the Writer Component Instance as a result of some other activity. During this execution, the Writer performs a publish Container Operation at **point 2**. The data is written to the Writer local copy of the repository at **point 3**. The ECOA Infrastructure is then responsible for:

- copying the data to any requiring Components (which may not be immediate depending upon the implementation of the Infrastructure).
- Generating then a notification Event for each Reader (at **point 4).**

The notification event is queued on the Reader Component Instance Queue at **point 5** as long as it is in the **RUNNING** state. This invokes the Reader notification Operation.

The Reader may then use standard Container Operations for getting a read-only copy of the latest data value (at **point 6**), as detailed in section 7.2.3.1.

Note: the latest data value at the time of the read request may be different from the data value at the time of the notification (in case of a data update in the meantime).

In case of a notifying Versioned data without access control, the Infrastucture generates an event to invoke Readers the same way, but writing and reading access apply on a single shared version of the data (with no management of local copies by the ECOA Infrastructure), as explained in section 7.2.3.6.

7.3 Component mechanisms

7.3.1 Component Instance Queues

Component Operation calls are placed in the Component Instance Queue, and the corresponding entry-point for the Component Instance is invoked when the Operation reaches the front of the queue (if the Operation is specified as an activating Operation, see section 7.9.2 for further detail on activating and non-activating Operations).

Component Operation calls, other than Component Lifecycle Operations, are only queued if the Component Instance is in the **RUNNING** state and if, for a particular Component Operation, the maximum number of waiting operation calls does not reach the value given by the attribute fifoSize defined on the receiving part of the associated OperationLink.

If the Component Instance is not in the **RUNNING** state Component Operations are discarded.

Whenever a request is discarded by the ECOA Infrastructure, the ECOA Infrastructure returns with « no response » error code to the Client Component as far as is possible.

Note: it is recommended that a timeout is specified on the Client side. This is due to the fact that some failures cannot be detected and a ‘no response’ error may never be received. For example, a network failure may prevent a remote ECOA Infrastructure on the server side from informing the client that it has discarded the request, or the request itself may never reach the server.

7.3.2 Property

Components may be instantiated multiple times within an Application. Component Properties provide a means to tailor the behaviour of a Component Instance.

A Component Property is declared as part of the Component Type. The associated value for a Component Instance is defined in the higher level Composite Assembly (see section 7.4.1).

A Component property value can be accessed at runtime, via the Container Interface, using a dedicated API in the Component implementation. The Architecture Specification Part 4 contains more detail regarding Properties.

7.3.3 Trigger

When a Trigger is defined within a Component, the ECOA Infrastructure manages services allowing each Instance of this Component to dynamically set and cancel alarms at runtime for its Trigger.

A Component can own several Triggers.

A Trigger is strictly local to a Component.

Each Trigger is associated to a no-parameter EventReceived in the Component Type.

An alarm is set with a delay which starts when the Trigger setting API is called by the Component Instance. As soon as the delay expires, the ECOA Infrastructure generates an Event to invoke the Component Instance, in accordance with the Event associated to the Trigger in the Component Type. This Event is always activating for the Component Instance (see 7.9.2).

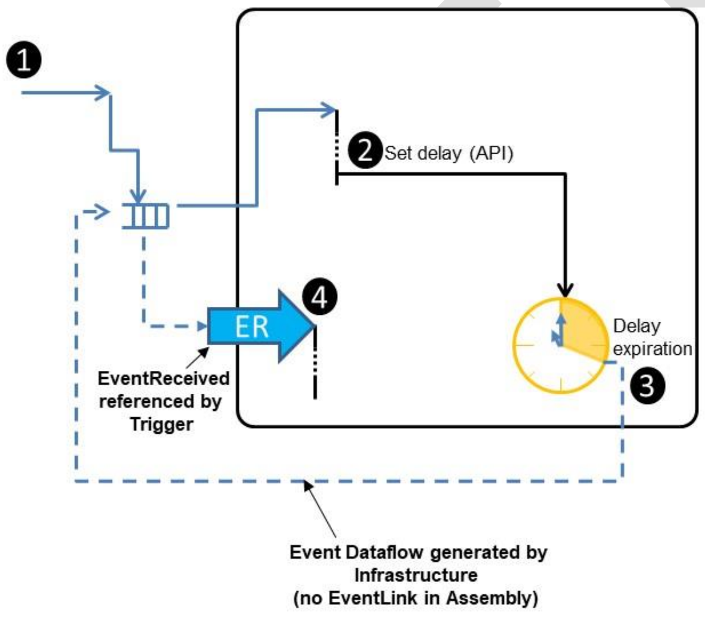

1
2
Set delay (API)
4
ER
EventReceived referenced by Trigger
Delay expiration
3
Event Dataflow generated by Infrastructure (no EventLink in Assembly)

Figure 13 Component Trigger behaviour

Figure 13 shows an example of a Component Instance which uses a Trigger to start specific treatments. It is supposed that the Trigger is not already activated. At **point 1**, an Operation is invoked on the Component Instance as a result of some other activity. During this execution, the Component Instance starts the Trigger with a specified delay (at **point 2**), so as to be awakened when this delay expires at **point 3**. So, the ECOA Infrastructure generates an event that is queued on the Component Instance Queue as long as it is in the **RUNNING** state. Specific treatments are invoked at **point 4**, according to scheduling rules of the platform (see section 7.9).

A Component which owns a Trigger can also cancel a previously activated Triggers.

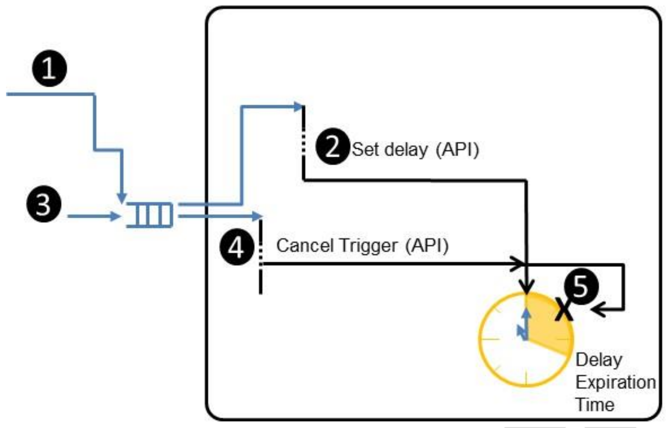


Figure 14 Component Trigger cancelled

Figure 14 shows an example of a Component which cancel a Trigger. At **point 1**, an Operation is invoked on the Component Instance as a result of some other activity. During this execution, the Component Instance starts the Trigger with a specified delay (at **point 2**), so as to be awakened when this delay expires. At **point 3**, an Operation is invoked on the Component Instance as a result of some other activity. During this execution, at **point 4**, the Component Instance cancel the Trigger previously started. Trigger is stopped at **point 5**: thus, expiration time will never be reached, and the associated Event will never be sent to the Component Instance.

Canceling a Trigger which is not started yet has no effect.

7.3.4 Persistent Information

7.3.4.1 Basic mechanisms

PINFO (Persistent Information) is a way of accessing, in read-only mode, to persistent information as if it were stored locally in a file through read and seek APIs.

A Component Instance can access to a PINFO if the following conditions are all satisfied:

- The PINFO is declared in the corresponding Component Type (using optional pinfo item),
- The association between PINFO name and PINFO value is described in the Assembly Schema. The name of a PINFO must be unique in the scope of a ComponentType.
- PINFO value corresponds to an actual accessible and available memory resource.

Figure 15 illustrates the aspects of PINFO and how they are being declared in an ECOA system.

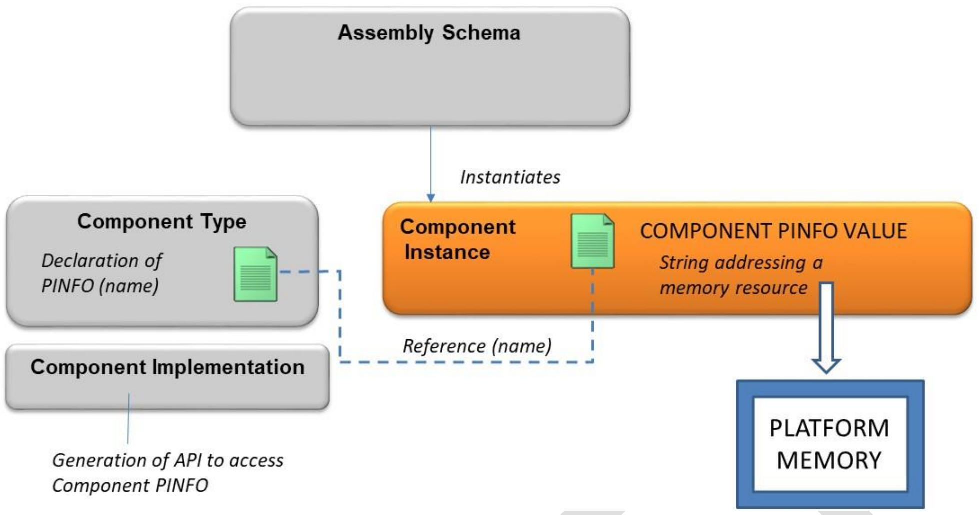


Figure 15 Aspects of PINFO

7.3.4.1.1 PINFO Attributes

PINFO have the following attributes associated with them:

- PINFO name:
  - This is the name used at Component level for accessing the PINFO. It is a logical name which has to be unique in the Assembly Schema.
- PINFO value:
  - This is a string allowing to address the Execution Platform Memory resource the PINFO actually represents. Depending on conventions defined for the Platform use, it can be for example a path and a filename, or a memory address etc.

    Writing rules for PINFO values depends on the Execution Platform specifications (this is not part of the ECOA Architecture Specification).

7.3.4.1.2 PINFO management

PINFO value is a post-production parameter: it can be modified after ECOA Software Platform building, with no impact on Application generated binary files.

The actual accessibility and availability of memory resources at defined addresses in PINFO value fields are not the responsibility of ECOA Infrastructure: they are the responsibility of the System Integrator, which has to ensure them prior to execution on the target ECOA Platform.

Considering the read-only access mode on PINFO, the ECOA Infrastructure does not manage any control on PINFO access.

7.3.4.1.3 PINFO API

The ECOA Infrastructure is responsible for making PINFO accessible to Component Instances:

- PINFO used by a Component Instance should be made accessible to that Component Instance prior to the INITIALIZE operation being invoked when it is in the IDLE state,

- PINFO used by a Component Instance should cease to be accessible to that Component Instance when it goes to the ‘IDLE’ state and any ongoing SHUTDOWN operation has returned.

From a Component Instance point of view, PINFO has the following form, illustrated by Figure 16:

- PINFO is an array of bytes,
  - Note: the use of PINFO may limit Components portability if the binary PINFO data and the functional code that accesses it are not designed to explicitly handle endianness in a platform independent way (i.e. byte order when accessing PINFO). It is up to Component suppliers to ensure that binary PINFO data is managed in a platform independent way.
- PINFO has an index, denoted by PINFO’index which indicates the current position in the array of bytes at which the next access will occur:
  - PINFO’index is expressed as a number of bytes,
  - PINFO’index starts from zero.
- PINFO has a size denoted by PINFO'size.

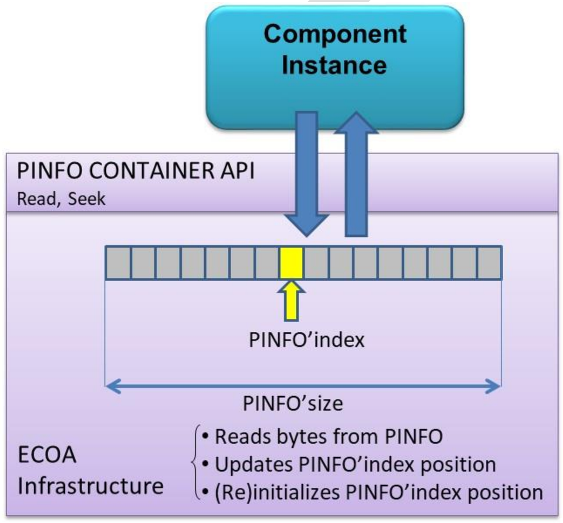


Figure 16 PINFO API

Component Instances invoke the following Container API for accessing PINFO:

- PINFO Read:
  - The Component Instance asks its Container to read a number of consecutive bytes in the PINFO, starting at PINFO’index position.
  - The Container provides the bytes read to the Component Instance.
  - The Container ensures that PINFO'size is not exceeded.
  - The Container increments PINFO’index according to the number of read bytes.
- PINFO Seek:
  - The Component Instance asks its Container to move PINFO’index forward or backward by a specified number of bytes relative to a starting position.
  - The starting position is one of ‘start of PINFO’, ‘current index’, ‘size of PINFO’.
  - The Container moves PINFO’index and returns its new position to the Component Instance.
  - The Container ensures that the new PINFO’index position is greater than or equal to zero.
  - The Container ensures that PINFO'size is not exceeded.

The ECOA Infrastructure sets the PINFO'index for a Component Instance to zero just prior to it invoking an INITIALIZE Component operation.

7.3.4.2 Writable PINFO [OPTIONAL]

Optionally an ECOA architecture may define PINFO in writable mode. This optional ECOA feature is not described in this document (cf. Part 6 - [OPTION PINFO WRITE]).

7.3.5 Specific kinds of component type

There are specific mechanisms related to kinds of Component Type which differ from the Standard one.

7.3.5.1 PeriodicTriggerManager

PeriodicTriggerManager Component allows to generate one or several periodic no-parameter Events, which are used to invoke other Component Instances within an Application. In the Assembly model, those Events are managed as usual (see 7.2.1), and may be linked with any Event received Operation that has no parameter.

PeriodicTriggerManager Component generated Events are queued to different Components of an Application e.g. where a single central PeriodicTriggerManager is used to coordinate the execution of treatments of multiple Components, in the manner of a "central clock".

The PeriodicTriggerManager Component is provided by the underlying Infrastructure to generate Events according to set time intervals called periods. Generated Events periods can be synchronised to the same start time, or synchronized on a delayed start time.

As any Component, the PeriodicTriggerManager Component exhibits Lifecycle Operations in its interface to enable the ECOA infrastructure to control its Lifecycle.

A PeriodicTriggerManager Component Instance is fully managed by the Infrastructure. Its implementation cannot be manually accessed.

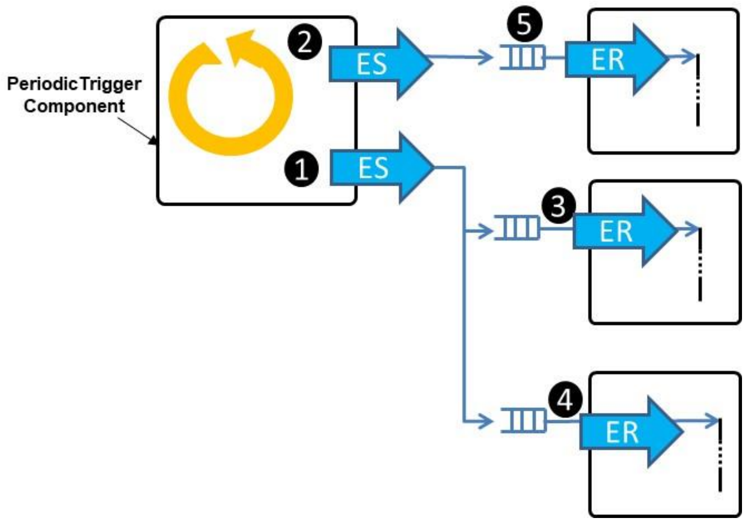


Figure 17 PeriodicTriggerManager Component

Figure 17 shows an Instance of a PeriodicTriggerManager Component which manages two periodic triggers. At **point 1**, it sends an occurrence of a trigger Event to two Components Instances, which are respectively activated at **point 3** and **point 4** (not necessarily synchronized, depending on communication latencies, rank of trigger Event in each Component Instance queue, executing entry-points at the time of trigger Event receipt etc.). At **point 2**, it sends an occurrence of another trigger Event to a third Component Instance which is activated at **point 5**.

7.3.5.2 External

External Components are a special kind of components, specifically designed to handle interactions between the ECOA and non-ECOA domains. This interaction may be with hardware devices or with non-ECOA software.

Regarding the interaction with other ECOA components (i.e. in component type and assembly models), an External Component is not different from a Standard Component. It is connected following regular rules, i.e. using Operations as described in section 7.2.

Regarding its internal implementation, an External Component has the following specificity: it has a dedicated additional thread, called the “external” thread, whose purpose is to allow using blocking calls.

Thus, contrary to other components, that respect the mono-threading rule, an external component therefore uses two threads:

- The "basic" thread, which executes all of the component's entry-points (related to operations and lifecycle), following the Inversion of Control principle. As with any component, this thread is not dedicated to the component; it may be shared with other components.
- The "external" thread, which is dedicated to the component.

Rationale

As exposed in Part 1, a Driver component is a component that handles interactions between the ECOA and non-ECOA domains. The most common problem when implementing a Driver component is that some form of "blocking wait" is required (for example, waiting for the reception of a message). This may imply, for example, calling a non-ECOA API that blocks the current thread for an unknown amount of time.

If a Standard component makes such a blocking call, it cannot do anything (receive events, etc.) until the unblocking of this call. Furthermore, it may block other components that share the same thread. The "external" thread allows doing blocking calls without these problems.

Thus, the External Component mechanism offers a solution for ECOA Platforms to support the implementation of the concept of Driver Components.

Like all Driver Components, External Components are generally less portable than other ECOA components, because they use at least one non-ECOA API. However, an external component can also be fully portable if it does not use such API.

The external thread

The external thread is automatically created by the infrastructure, but its implementation is left to the component itself.

There can be several External Components in an Application. Each instance of External Component has its own, independent, external thread.

The priority of the external thread is linked to the priority of the component instance (which is the priority of the basic thread). The external thread has a lower priority than the basic thread (so that the basic thread can control the external thread), but higher priority than all other components of lower priority.

The external thread can be started or stopped on the ECOA treatments request; it is also possible to start it automatically, through option "autostartExternalThread" of the implementation model. (this option is available only if the ECOA platform support [OPTION AUTO START EXTERNAL THREAD]).

Code executed by the external thread is allowed to call the component container API.

It is possible to send events, send asynchronous requests, read and write data from the external thread.

It is not possible to use a synchronous request-response from the external thread.

Synchronisation between threads

An External Component is in charge of the communication and synchronisation between its code executed by the basic thread, and its code executed by the external thread.

In some cases, no synchronisation is necessary (for example, if the main thread does nothing).

If private data from the user context is used from both threads, then some synchronisation is needed to avoid data inconsistency.

The following mechanisms may be used for this synchronisation:

- ECOA data operations,
- ECOA operations (e.g. sending an event from the external thread and receiving it on the basic thread),
- atomic variables in the user context,
- lock-free data structures,
- use of low-level synchronisation services provided by the OS (mutexes, semaphores),
- etc.

7.3.5.3 Supervisor

A **SUPERVISOR** component type is the only type of component that will have the knowledge of the current Application's assembly and deployment. This component can access the list of component instances of the Application, access to the state of all component instances, and control it (initialize, start, stop, shutdown cf. Component state diagram). It can also start and stop the Executables of the Application. This feature is not supported by all platforms, it is part of [OPTION SUPERVISION].

See Optional Part related to [OPTION SUPERVISION].

7.3.5.4 DynamicTriggerManager

An optional feature allows to define DynamicTriggerManager Components ([OPTION DYNAMIC TRIGGER]) in order to generate Components able to trigger others at a configurable absolute time.

See Optional Part related to [OPTION DYNAMIC TRIGGER].

7.4 Composite and Assembly

Composites enables a recursive definition of ECOA Software Architectures, to offer hierarchical views on them and by this way, ease understanding.

As any Component, a Composite Component is defined by a Component Type, but the Composite implementation is an Assembly instead of a piece of software, as it only consists in a set of Components (which can themselves be Composites) and a set of Operation Links.

A composite Component has no existence at runtime.

A Composite Component has no Lifecycle nor Trigger.

The assembly associated to a Composite allows to define:

- Components Instances owned by the Composite,
- Values of Component Instance properties and PINFO,
- Links between Component Instances (which might include Conditional Links if [OPTION SUPERVISION] is available),
- Promotion Links between Component Instance operations and Composite external operations.

7.4.1 PINFO and Properties in Composite Components

It is possible to define properties or PINFO in the Component Type of a Composite Component: it allows to define common values for all Components instantiated within the Composite (see 7.3.1 for Property and 7.3.4 for PINFO).

Note that in order for a Composite Property/PINFO to be accessed, a Property/PINFO referencing this Composite Property/PINFO must be created in the Component Type of a Component instantiated within the Composite.

Within the Assembly Schema the Component Instance Property/PINFO Values are assigned for each Component Instance. These values are assigned at design time and cannot be changed during execution.

A Component Instance Property/PINFO Value may either be a literal value or a reference to an Assembly Property Value.

An Assembly Property/PINFO Value is a property/PINFO which is accessible by all Components within the Assembly.

See Architecture Specification Part 7 for more details.

7.4.2 Composite internal assembly

A Component or an Application, connected to a Composite external Operation, has no knowledge of the internal Composite Components.

From the Infrastructure point of view:

- Within an Application: if a Component is connected to an external Operation of a Composite, it has exactly the same effect than if the Component was directly connected to the internal Component Operation that is promotionally linked to the Composite external Operation.
- For an Application level Composite: external Operations cannot be directly connected to external Applications. External Communications are only possible through Application Ports (see section 7.5.2).

Figure 18 shows an example Composite constructed with four Components.

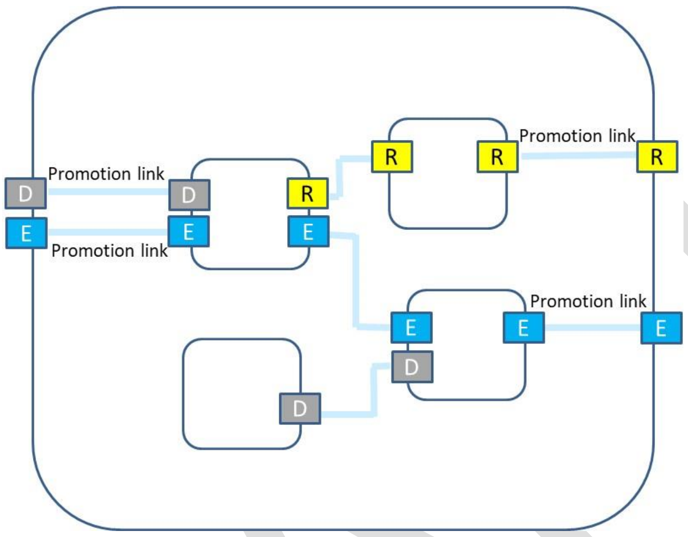


Figure 18 A Composite

7.5 Application characteristics

7.5.1 Assembly, Component and Properties

An Application is associated to the highest level of Composite. Then requirements defined in section 7.4 fully apply.

7.5.2 Application ports

This feature is not supported by all platforms, it is part of [OPTION COMM PORTS].

Application ports are artefacts allowing to define the Application interface, regarding software considerations (typically, transport protocols needs).

Ports are groups of Operations, that will be connectable to another Application. The entire list of Operations referenced by the Port will be connected when the Port itself will be connected.

Applications ports can be either mono-directional (In Ports, Out Ports) or bi-directional (InOut Ports).

ECOA internal communication mechanisms (Event, Versioned Data, RequestReponse) are no more available for interactions between Applications.

7.6 Interactions outside of the Application

The management of interactions between ECOA Applications is not a required feature of ECOA Architecture Specification.

It nevertheless remains possible to address the global topic of (ECOA or non-ECOA) software interactions, using other technologies dedicated to distributed systems (such as gRPC for example).

ECOA Extensions may be defined to enable the management of a distributed ECOA system. This may be particularly interesting for sub-systems entirely based on ECOA Applications.

To communicate outside of the Application:

- If interactions remain compliant to ECOA communication mechanisms (Event, Versioned Data, Request-Response), interactions may be implemented by connecting Applications Ports . The use of Application Ports in software interactions requires the definition of a binding to a communication protocol, allowing to specify how Ports and associated Operations are implemented according to this protocol mechanisms.

- Else: it is possible to rely on External Component mechanisms (see section 7.3.5.2).

Note : when an ECOA Platform supports [OPTION EXTERNAL INTERFACE], an external interface is also defined to allow non-ECOA software to asynchronously interact with one or more Components (see Architecture Specification Part 6).

7.7 Infrastructure services for Application management

7.7.1 Lifecycle management

7.7.1.1 Components Startup

Following allocation and initialization of resources by the Infrastructure each Component Instance is brought from **UNAVAILABLE** state to the **IDLE** state.

For Component Instances who have reached the **IDLE** state, the next steps of an Application start-up sequence can be automatic or not, depending on the start mode selected in the ECOA model:

- Start mode = none: ECOA Infrastructure does not automatically manage components INIT and START. When defined, SUPERVISOR components might manage other components start-up. Components might also be started using a GUI if available in the platform.

- Start mode = fast: with the help of Lifecycle Events, the ECOA Infrastructure requests each Component Instance to initialize (**INIT** transition) and to **START**. Components "INIT and START" sequences may be executed in any order or even in parallel.

- Start mode = synchronized:

  With the help of Lifecycle Events, the ECOA infrastructure first requests all Component Instances to initialize (**INIT** transition) in the following sequential order:
    - DynamicTrigger Components (if defined),
    - PeriodicTriggerManager Components,
    - Other components.

When all Component Instances of the Application are in **READY** state, the ECOA Infrastructure requests all Component Instances to **START** in the following sequential order:

- DynamicTrigger Components (if defined),
- PeriodicTriggerManager Components,
- Other components.

Note: Trigger Components should be started before STANDARD components because the overhead of starting them is more controllable.

In its **INIT** entry point, each Component Instance performs any actions required to initialize such as allocating further resources, setting its internal variables and/or reading its Properties to reach a technically coherent and initialized internal state.

In its **START** entry point, each Component Instance has the opportunity to perform any actions relevant to its transition to the **RUNNING** state (these are likely to be specific to the functionality of the design). Once started the Component Instance enters the **RUNNING** state.

The technical start-up phase is over when all types of Component Instances within the Application are in **RUNNING** state.

Note: further functional initialization of the ECOA System may take place once Component Instances have reached the RUNNING state (however from the lifecycle point of view, the technical initialization of Component Instances is considered to be complete at this stage).

The following are the steps taken to change the state of a Component Instance that is not **UNAVAILABLE**:

- A SUPERVISOR component or the ECOA Infrastructure requests a transition to be applied to the Component Instance lifecycle,
- the Container invokes the appropriate entry point in the Component Interface
- when the entry point returns the Container changes the state of the Component Instance

7.7.1.2 Component Run-time Behaviour

Lifecycle Events that do not represent a valid transition from the current state, as shown by the Component Lifecycle state diagram in Figure 2, are discarded, and the optional fault management service may be notified (see section 7.7.3).

Lifecycle Events are activating operations.

Lifecycle Events have no priority over other operations:

- Operations arriving before the **START** entry-point has returned are discarded, as illustrated in Figure 19.

- The ECOA Infrastructure discards any operation (except legal Lifecycle Events) arriving to a Component Instance that is not in the **RUNNING** state (i.e the operation is not queued), as illustrated on Figure 19 and Figure 20.

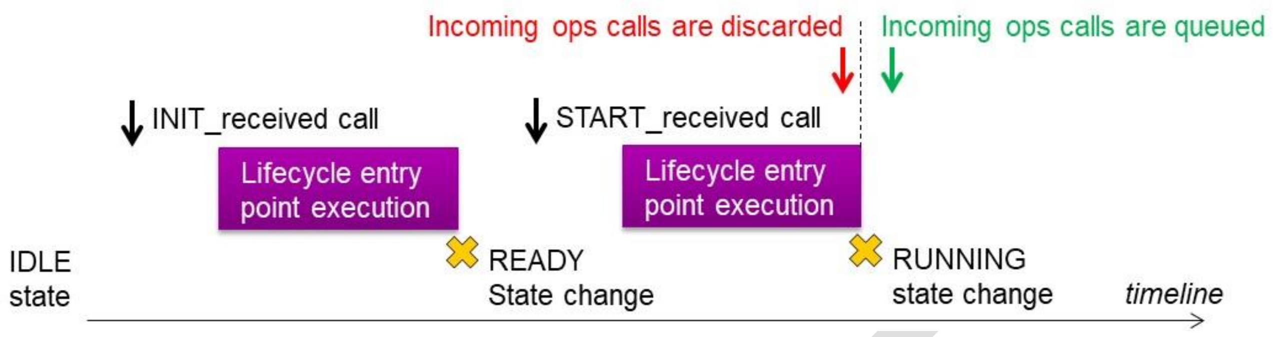

Incoming ops calls are discarded
Incoming ops calls are queued
INIT_received call
START_received call
Lifecycle entry point execution
Lifecycle entry point execution
IDLE state
READY State change
RUNNING state change
timeline

Figure 19 Component Run-time Behaviour – Startup

- On receipt of **STOP** Events, operation entry points already executing are allowed to complete and operation calls may continue to be queued but are finally discarded as soon as the **STOP** or **SHUTDOWN** entry point has been executed (i.e. once the Component Instance is not in **RUNNING** state anymore, the Component queue is cleared).

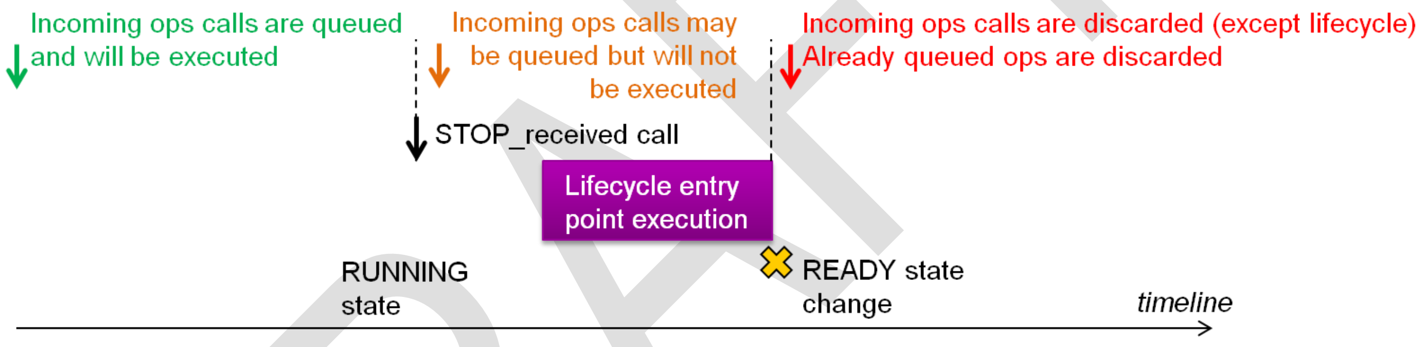

Incoming ops calls are queued and will be executed
Incoming ops calls may be queued but will not be executed
Incoming ops calls are discarded (except lifecycle)
Already queued ops are discarded
STOP_received call
Lifecycle entry point execution
RUNNING state
READY state change
timeline

Figure 20 Component Run-time Behaviour – Stop

- Operations arriving after receipt of **SHUTDOWN** Events (whilst in **RUNNING**) may be queued, but are finally discarded as soon as the **SHUTDOWN** entry point has been executed (i.e. once the Component is not in **RUNNING** state anymore, the Component queue is cleared), as illustrated in Figure 21.

- Operations arriving whilst in **READY** continue to be discarded and the **SHUTDOWN** Event does not change this, as illustrated in Figure 21.

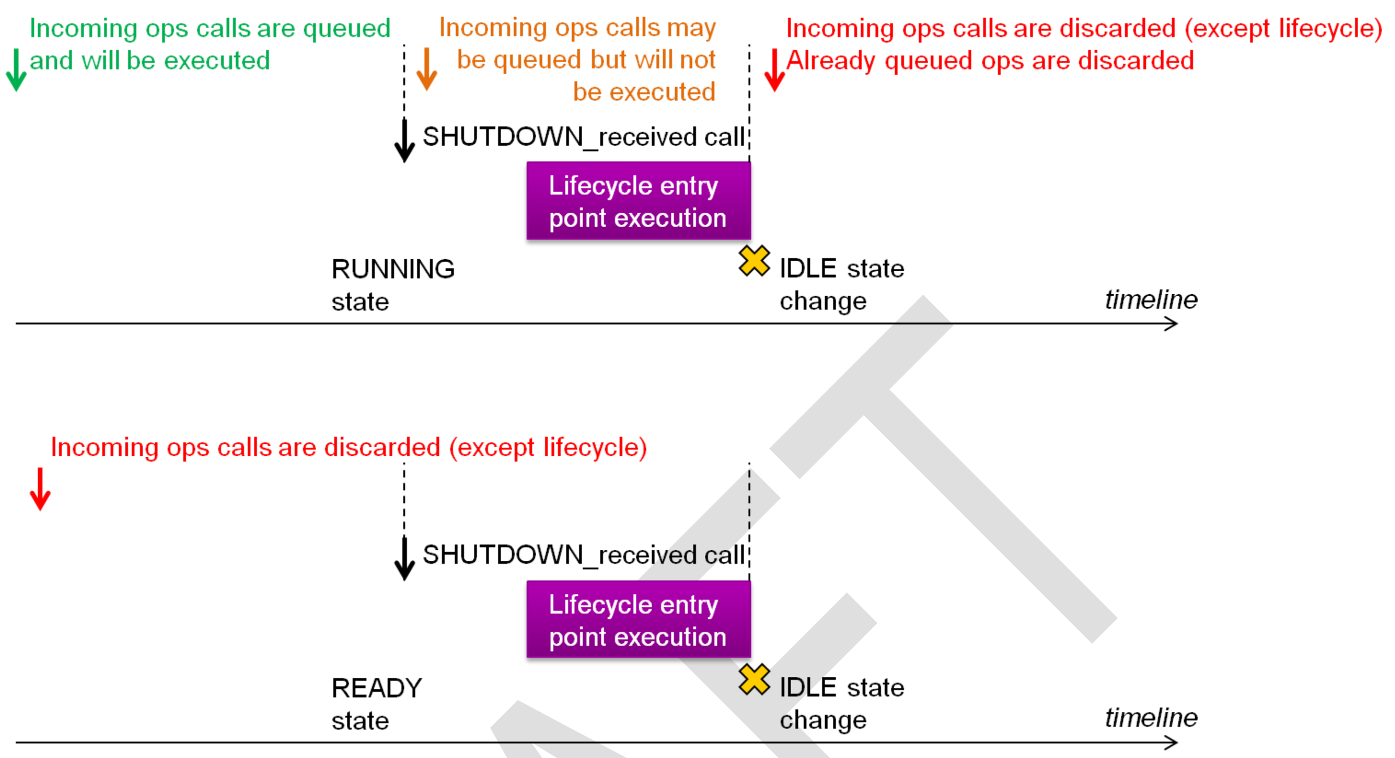

Incoming ops calls are queued and will be executed
Incoming ops calls may be queued but will not be executed
Incoming ops calls are discarded (except lifecycle) Already queued ops are discarded
SHUTDOWN_received call
Lifecycle entry point execution
RUNNING state
IDLE state change
timeline
Incoming ops calls are discarded (except lifecycle)
SHUTDOWN_received call
Lifecycle entry point execution
READY state
IDLE state change
timeline

Figure 21 Component Run-time Behaviour – Shutdown

A Component should not invoke any Request-Response operations in its Container Interface as part of the implementation of a Component Lifecycle operation. The Component may however invoke Event or Versioned Data operations in this case.

7.7.1.3 Component Shutdown

When its **SHUTDOWN** entry point is called, a Component Instance must de-allocate any associated resources (those previously allocated in **INIT** entry point).

When the **SHUTDOWN** entry point of a Component Instance returns, the Infrastructure shall de-allocate the resources used by the Component Instance (those previously allocated by Infrastructure in Component Instance INIT entry point), after which the Component Instance enters the **IDLE** state.

Whenever a Component Instance enters the **IDLE** state, its queue is cleared and the Infrastructure releases all Versioned Data handles associated with it.

7.7.1.4 Graceful vs fast shutdown

7.7.1.4.1 Graceful shutdown procedure

The ECOA Infrastructure may call the **STOP** and **SHUTDOWN** entry points of Component Instances for the purpose of performing a "graceful shutdown" procedure.

A graceful shutdown may be useful for simulation purposes in order to be able to stop, shutdown and restart selected Components and let them perform user-specified code as part of these entry points. It may also be useful for EXTERNAL components so that these can free any specific underlying resource before becoming **IDLE**.

ECOA itself does not mandate ECOA Platforms to be capable of graceful shutdown. In order for a graceful shutdown procedure to be possible, the capability for it needs to be procured within the ECOA Software Platform. In addition, the functional specifications of targeted Components need to tell the Component supplier what application code to implement in the **STOP** and **SHUTDOWN** entry points.

7.7.1.4.2 Fast shutdown procedure

As part of some recovery actions, the ECOA Infrastructure may perform a “fast shutdown” procedure of Component Instances, by putting them into **IDLE** state without invoking their **STOP** or **SHUTDOWN** entry points.

7.7.2 Health Monitoring

Unlike Fault Handling, ECOA assumes that Health Monitoring is related to Application and functional design and that it can be addressed by designing Components for this purpose.

7.7.3 Fault Handling

7.7.3.1 Error Categorization

Error is a global term that encompasses:

- Application errors: consequences of faults occurring at Component level. Their management is the responsibility of the component provider. Such an Error can be:
    - A fatal Error: the Component knows it cannot recover on its own and uses mechanisms provided by the Infrastructure to report the Error
    - A non-fatal Error: the Component may be able to recover on its own (in case of minor Errors) or it may report the Error
- Infrastructure errors: consequences of faults occurring at ECOA Infrastructure level. Their management is the responsibility of the system architect and system integrator.

7.7.3.2 Error Propagation and Recovery Actions

Errors can be detected at several levels:

- Component level (application errors)
- Component Container level (Infrastructure errors)
- Executable level (Infrastructure errors)

Fault management may provide recovery procedures at:

- Component level
- Executable level

Detection and recovery procedures for Infrastructure errors may be specified in more details in extensions of the ECOA Architecture Specification.

In any case, the platform provider shall explain the platform fault management policy.

7.7.3.2.1 Errors detection and recovery actions

Application errors may be detected by a Component. The Component provider determines the recovery actions (if any) to be applied by the Component when an application error occurs.

If an application error is detected by a Component, the recovery procedure may be:

- In case of non fatal application error, the Component may:
    - Handle the application error if it has been coded to recover from that type of error,
    - Raise the application error to the Fault Handler (if available) through *raise_error*.
- In case of fatal application error, the Component uses the *raise_fatal_error* API.
    - The error is automatically sent to the Fault Handler (if available),
    - The faulty Component is immediately placed into the IDLE state by the ECOA Infrastructure.

Mechanisms related to Fault Management at application level are explained in Architecture Specification Part 6 ([OPTION FAULT HANDLER]).

Complementary Fault Management mechanisms at system level may be specified as an extension of the ECOA Architecture Specification.

7.7.4 Component User Context

When a Component is not stateless, its source code should save states using the User Context, and should not use global variables. This enables a platform to instantiate a single component implementation multiple times and manage instance specific data contained in the Component User Context.

User Context is instance specific data defined and managed by the Component Implementation to allow it to maintain private state between each component operation invocation. This is analogous to the context of thread local storage in traditional operating system implementations.

Note: the Component supplier shall initialize the User Context every time the Component Instance INIT entry point has been called, and shall never assume the User Context has been maintained by the Infrastructure.

The User Context is always volatile, whatever the type of platform startup (cold start or warm start).

Architecture Specification Part 4 describes the detail of how the User Context is represented in various languages, and the language bindings specify the detail of the implementation.

7.7.5 Component Warm Start Context

For Components having an Implementation where the attribute `hasWarmStartContext` is set to true, an API is generated for related instances so that they might:

- declare the structure of a warm start recovery context,
- save a warm start recovery context.

Warm Start Context saving and restoring mechanisms are not supported by all platforms ([OPTION WARM START CONTEXT]). They are not described in this version of the document

7.8 Building and deploying

7.8.1 Generation and building

Source code generation shall respect language specific constraints specified by Component Implementation elements related to file prefixes and package names.

Component Instance Containers shall implement the ECOA API referenced by attributes `APIType` and `APIVersion` specified in Component Implementations (supposing all of them are available in the Software Platform).

The ECOA Infrastructure should respect building instructions defined in:

- Component Implementation attributes (Extra elements: source code directories, compilation flags, etc.).
- Deployment attributes (production, endianness applied to the physical network)

See Architecture Specification Part 5 for details.

7.8.2 Deployment

Any ECOA Software Platform shall respect the process and thread software architecture defined by the Software Architect in the Component Instances deployment, which means the Application is a main process that creates a child subprocess for each sub-Executable defined in the Application Deployment.

Any ECOA Software Platform shall actually deploy Component Instances onto OS threads following these rules:

- Two tasks of the same Executable are deployed onto two different OS threads of the same process,
- The list of Component Instances deployed onto an OS thread is the same than the one specified for the corresponding Task,
- When several Component Instances deployed onto the same OS thread are activated by the same Operation, they are activated in the same order as the one of the list of deployed Component Instances associated to the corresponding Task.

Upon start-up of an ECOA Software Platform, the ECOA Infrastructure is responsible for the allocation and initialization of all the resources (threads, libraries, objects, etc.) needed to execute the functionality of the Component Instances and their Containers for each Application.

Memory allocations take into account Component Implementation attributes `stack` and `externalStack`.

7.9 Scheduling

7.9.1 Scheduling Policy

Support for scheduling of deployed Component Instances is provided by the underlying operating system. Except for EXTERNAL Components, a deployed Component Instance is single-threaded and inherits the priority of its associated Task, as assigned by the Software Architect.

Note: EXTERNAL Components additional thread is automatically assigned the next higher priority than its associated Task.

Any ECOA Software Platform shall respect the priorities defined by the Software Architect on Tasks. This implies:

- Deploying Executable Tasks onto OS threads according to their priorities,
- Task pre-emption capability in ECOA platforms:
  - Any Task becomes eligible for execution on the computing node resource whenever one of its associated Component Instance receives an activating operation in its queue (including reception of the response of a synchronous request-response),
  - A Task currently executing on the computing node resource may be pre-empted by another eligible Task having a higher priority,

  - A Task becomes ineligible for execution on the computing node resource when none of its associated Component Instances has any activating operation in its queue.

The rules defined above allow early verification of the system behaviour to take place at ECOA Component Instance level, and provide assumptions to facilitate Component integration and reuse.

Scheduling of OS threads (which Component Instances are being deployed onto) is the responsibility of the Infrastructure and any scheduling policy supported by the OS/Middleware may be used as required by the System Integrator, provided that it respects the rules defined above. Scheduling analysis is outside the scope of ECOA; although it is anticipated that it would be carried out following existing, established methods. The type of scheduling analysis required will be dependent upon the chosen scheduling policy.

7.9.2 Activating and non-Activating Component Operations

By default, Operations are activating; the arrival of a new operation implies the execution of the associated entry point as soon as the Component Instance is able to execute. This schema is an Event-driven programming model.

To disable this default behaviour, attributes are defined within the Component Implementation at `EventLink`, `RequestLink` and `DataLink` level:

- `activating` which is a boolean specifying the policy used by the Container to handle the operation:
  - when True (default value), the Container activates the associated entry point as soon as possible.
  - when False, the operation is queued and remains pending. When an activating Operation arrives to the same Component Instance through another Operation Link, all pending Operations that arrived before the activating one are then processed in FIFO order and executed as any other Operation in accordance with the priority of the Component Instance. If non activating Operations are queued while this processing is done, their processing is postponed until the arrival of a new activating Operation. It is envisaged that this type of mechanism could be used to implement a time-driven programming model, which may allow for easier schedule feasibility analysis.
- `fifoSize` which is an integer specifying the maximum number of pending operations of a single type at each receiving Container level. The `fifoSize` attribute is defined against an operation Link; therefore different operations can be specified to have different maximum queue sizes. When the maximum number is reached for a given operation, the receiving Container discards the new incoming Operations associated to the Link. For an incoming Request Response, the receiver Container sends back an error message to the sending Container in order to notify the Client of the failure.

  Note that `fifoSize` attribute is not available on Implicit Links (so it is set to default value).

However, this choice does not apply to Component Lifecycle Operations, which are necessarily activating operations (i.e. it is not possible to define them as being non-activating).

NOTE: Activating on DataLink is only useful when associated to a notifying Versioned Data (attribute `notifying` set to `true`) (see §7.2.3.7).

7.10 Utilities

An ECOA Software Platform provides utility functions for acquiring time and for generating logs. The Architecture Specification Part 4 contains more detail regarding these functions.

Some of the ECOA Software Platform provided functions are methods for allowing access to time. These methods are available in Component Containers as soon as Option elements related to local, system or UTC Time are selected.

Local time is always available on an ECOA platform.

When a synchronised system time is not available, local time is returned by the ECOA Infrastructure on system time request (with an error code to warn the calling Component Instance).

Note : the availability of a system time may be addressed by an extension of ECOA Architecture Specification.

UTC time is not supported by all ECOA platforms ([OPTION UTC TIME]).

Component instances are offered different logging functions allowing to select the logging level to be used.

Requirements about logging mechanisms configuration are described in Architecture Specification Part 5.