European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 5: Mechanisms Reference Manual

BAE Ref No: IAWG-ECOA-TR-007
Dassault Ref No: DGT 144482-B

Issue: 2

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: *This specification represents the output of a research programme and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This standard of documentation is recommended as appropriate for limited lab-based evaluation only. Product development based on this standard of documentation is not recommended.*

1 Table of Contents

1 Table of Contents 2
2 List of Figures 4
3 List of Tables 5
4 Abbreviations 6
5 Introduction 7
6 ECOA Mechanisms 9
7 Interactions 10
7.1 Module Instance Queues 10
7.2 Event 11
7.2.1 Event Sent by Provider 11
7.2.2 Event Received by Provider 12
7.3 Request Response 12
7.3.1 Synchronous Request 13
7.3.2 Asynchronous Request 14
7.3.3 Immediate Response 14
7.3.4 Deferred Response 15
7.4 Versioned Data Publication 15
7.4.1 Notifying Versioned Data 17
7.5 Trigger 18
7.6 Dynamic Trigger 19
7.6.1 Dynamic Trigger Operations 20
7.6.2 Dynamic Trigger management 21
7.6.3 XML definitions of Dynamic Trigger Instance and associated links 21
7.7 Interactions within Components 22
7.8 Component and Module Properties 23
8 ECOA System Management 24
8.1 Lifecycle 24
8.1.1 Component Runtime Lifecycle 24
8.1.2 Module Runtime Lifecycle 26
8.1.3 Lifecycle Example 29
8.2 Health Monitoring 29
8.3 Fault Management 29
8.3.1 Fault Categorization 30
8.3.2 Fault Propagation 30
8.4 Run-time Configuration Management 31

8.4.1 Initialisation 31
8.4.2 Reconfiguration 31
9 Scheduling 32
9.1 Module Deadline 32
9.2 Scheduling Policy 32
9.3 Activating and non-Activating Module Operations 33
10 Service Availability 34
10.1 Initialisation 34
10.2 Assembly Schema 34
10.2.1 Service Links and Ranks 34
10.3 Dynamic Service Availability 35
11 Service Link Behaviour 36
11.1 Introduction 36
11.2 Active Provider Component 36
11.3 Summary of Behaviour 36
11.4 Examples 37
12 Module Operation Link Behaviour 42
13 Utilities 44
14 Inter Platform Interactions 45
15 Composites 46
16 References 47

2 List of Figures

Figure 1 – ECOA Documentation 7
Figure 2 – ECOA Interactions – Key 10
Figure 3 – Event Sent by Provider 11
Figure 4 – Event Received by Provider 12
Figure 5 – Synchronous Client Request-Response 13
Figure 6 – Asynchronous Client Request-Response 14
Figure 7 – Deferred Response Server Request-Response 15
Figure 8 – Versioned Data Behaviour 17
Figure 9 – Notifying Versioned Data Behaviour 18
Figure 10 – Trigger Behaviour 19
Figure 11 – Dynamic Trigger Behaviour 20
Figure 12 – Interactions within Components – Synchronous Request-Response 23
Figure 13 – Component-level lifecycle states 24
Figure 14 – Module Runtime Lifecycle 26
Figure 15 – Lifecycle Example 29
Figure 16 – Fault propagation in an ECOA System 30
Figure 17 – Service Links 36
Figure 18 – Example Assembly Schema 38
Figure 19 – Generation of an Event 38
Figure 20 – Consumption of an Event 39
Figure 21 – Synchronous Request-Response Operation 40
Figure 22 – Asynchronous Request-Response Operation 40
Figure 23 – Selection of Versioned Data 41
Figure 24 – Interactions between Service Operations and Module Operations 42
Figure 25 – A Composite 46

3 List of Tables

Table 1 – Behaviour across a Service Link 37
Table 2 – Table of ECOA references 47
Table 3 – Table of External References 48

4 Abbreviations

API	Application Programming Interface
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
FIFO	First In, First Out
QoS	Quality-of-Service
UDP	User Datagram Protocol

5 Introduction

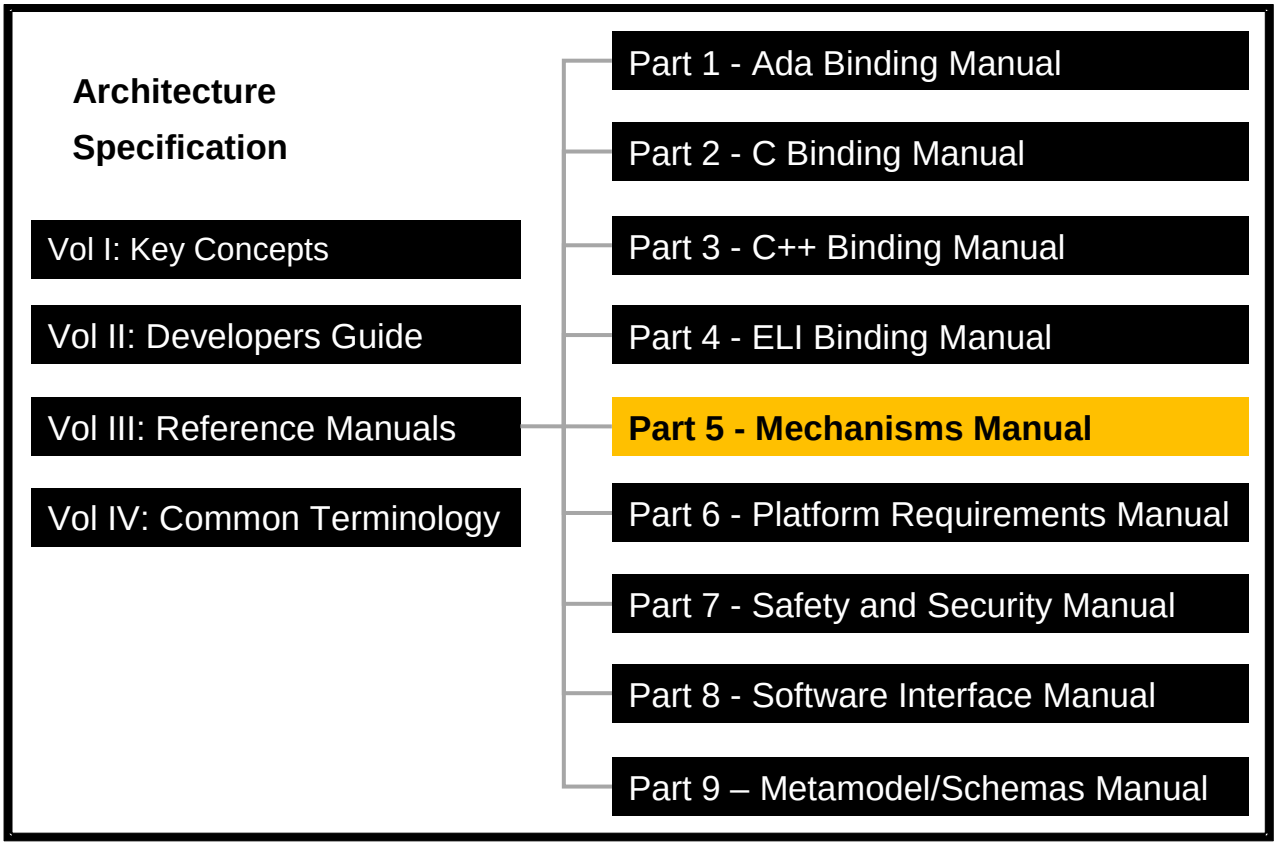

Figure 1 – ECOA Documentation

The Architecture Specification provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA-based system. It is introduced in Key Concepts (Reference 1) and uses terms defined in the Common Terminology (Reference 11). For this reason, the reader should read these documents, prior to this document. The details of the other documents comprising the rest of the Architecture Specification can be found in Section 16.

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

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

This document comprises Volume III Part 5 of the ECOA Architecture Specification, and describes the various mechanisms that are provided by an ECOA system.

The document is structured as follows:

- Section 6 provides an overview of the mechanisms that are used for interactions between Modules in the system.
- Section 7 describes in details the behaviour of the interactions in an ECOA system.

- Section 8 describes the System Management mechanisms that are provided by the Infrastructure.
- Section 9 describes the support for scheduling within an ECOA system.
- Section 10 describes the mechanisms for managing Service Availability in an ECOA system.
- Section 11 describes Service Link behaviour.
- Section 12 describes Module Operation behaviour.
- Section 13 describes the utilities provided by the ECOA Software Platform.
- Section 14 describes how inter-platform communication occurs within an ECOA system.
- Section 15 describes the concept of a composite (collection of Application Software Components)
- Section 16 contains details of other document referenced from this one.

6 ECOA Mechanisms

The ECOA concept (Reference 1) defines an architecture which uses Application Software Components and Services. This document describes the mechanisms defined by the ECOA and the way that Components interact. In addition it describes the behaviour of other aspects of an ECOA system including management and utility functions along with how different ECOA Software Platforms interact.

Some of the mechanisms are described in detail within this document, whereas others are only discussed at a high level, as they are covered in greater depth in other documents. Where this is the case a reference will be provided.

The intended audience for this reference manual is:

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

This document describes the mechanisms available to an Application Software Component, but it is the ECOA Software Interface Reference Manual (Reference 9) which provides the abstract API for implementing the mechanisms described herein.

7 Interactions

Interactions between Module Instances in an ECOA system rely on three primary mechanisms:

- Events
- Request-Response
- Versioned Data publication

The interactions between Module Instances can occur within a single Application Software Component, or between Module Instances of different Application Software Components, as a consequence of their Services. For detail on the behaviours, see sections 11 and 12 respectively.

In addition to the above mechanisms, Operations exist for Infrastructure Services to allow the management of the runtime lifecycle, Properties, logging, faults and time Services.

The following sections include numerous figures, which illustrate the interactions within an ECOA system and provide visual clarity. The key shown in Figure 2 offers guidance on the colouring and symbology used throughout these sections.

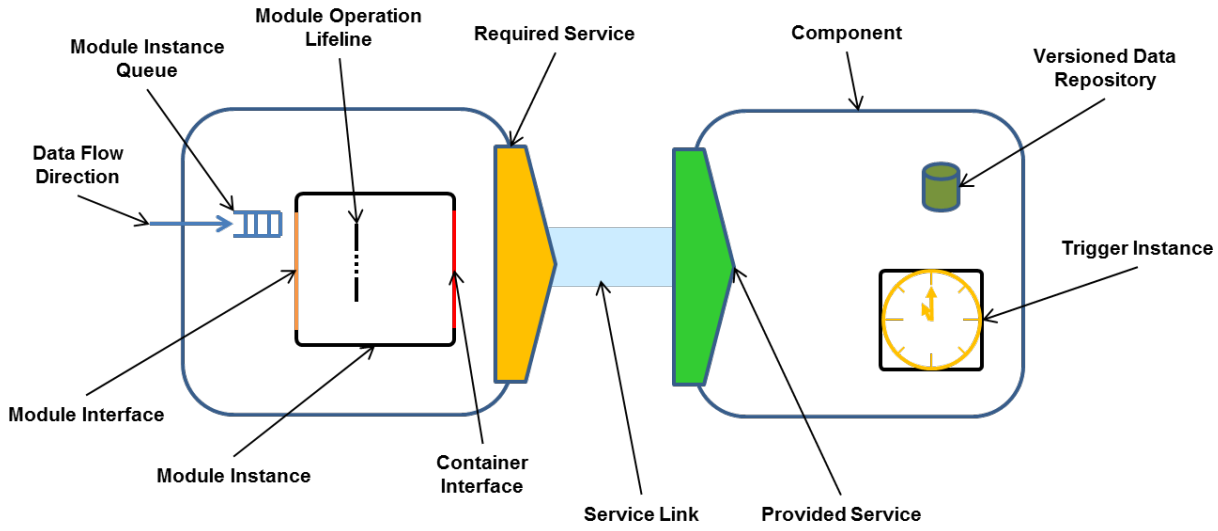

Figure 2 – ECOA Interactions – Key

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

7.1 Module Instance Queues

Module run-time behaviour is dependent upon the Module Runtime Lifecycle state (see section 8.1.2). A set of predefined Module Operation called Module Lifecycle Operations exist to allow the Container to inform the Module of changes to its Lifecycle. Module Lifecycle Operations are handled in any state (to enable the Lifecycle of a Module Instance to be managed), whereas normal Module Operations are only handled in the **RUNNING** state.

Module Operation calls are placed in the Module Instance Queue, and the corresponding entry-point for the Module Instance is called when the Operation reaches the front of the queue (if the Operation is specified as an activating Operation, see section 9.3 for further detail on activating and non-activating Operations).

Module Operation calls other than Module Lifecycle Operations are only queued if the Module Instance is in the **RUNNING** state.

If the Module Instance is not in the **RUNNING** state Module Operations are discarded. For Request operations arriving to a non-RUNNING Module, the Container will directly return a Response indicating that the Operation is not available.

7.2 Event

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

When Events are used to implement a Service Operation, a Module Instance may be either the sender or receiver of an Event irrespective of whether it is designated as the Provider or Requirer of the Service.

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

There may be multiple receivers of an Event within a Component (e.g. other Module Instances or Service Instances), in which case instances of the Event are broadcast to all receivers.

For Events between Component instances, the behaviour is defined by the Rank and allEventsMulticasted attributes associated with the Service Link. If the Service Link is not identified as allEventsMulticasted; then only the Component instance connected to the Service Link with the lowest value of Wire Rank shall receive the Event. Further detail of this behaviour is described in section 11.

7.2.1 Event Sent by Provider

In the case of an Event sent by Provider, the providing Application Software Component initiates the sending. This behaviour is shown in Figure 3.

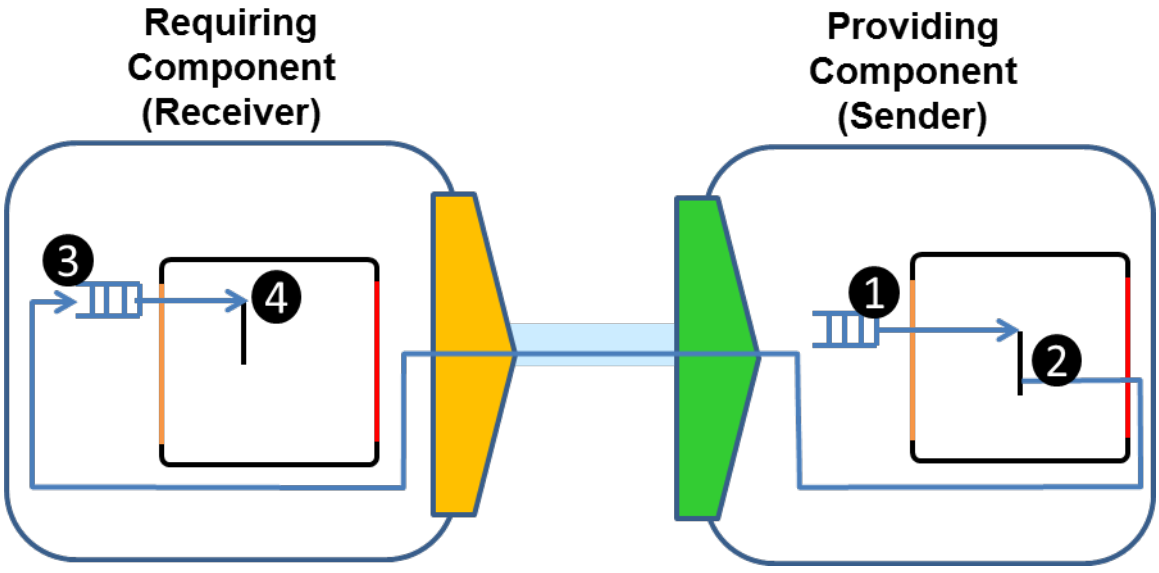

Figure 3 – Event Sent by Provider

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

7.2.2 Event Received by Provider

In the case of an Event received by Provider, the requiring Component initiates the sending. This behaviour is shown in Figure 4.

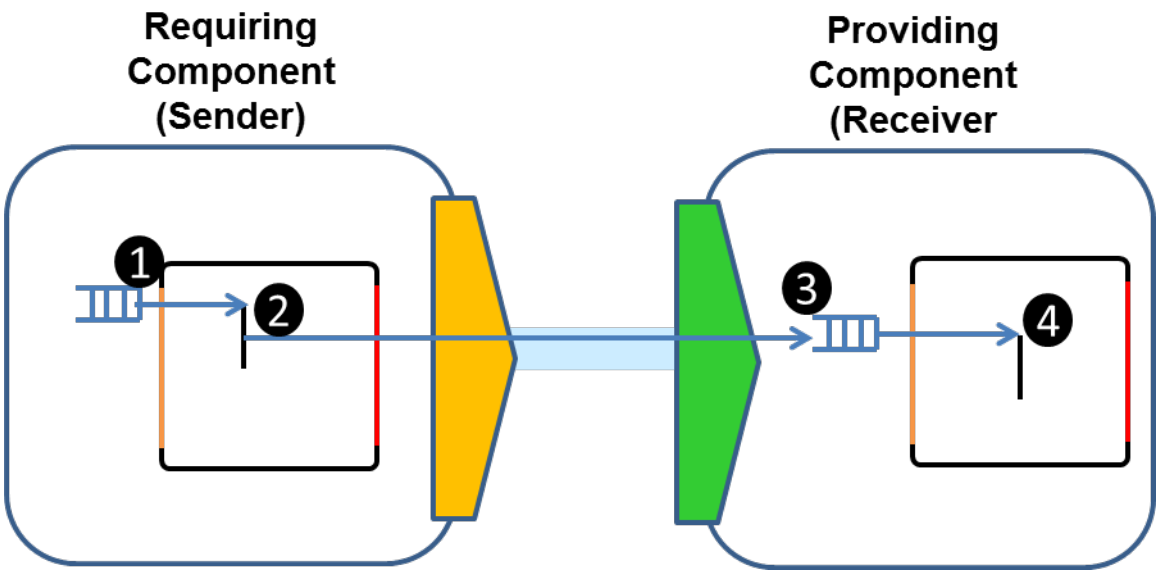

Figure 4 – Event Received by Provider

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

7.3 Request Response

The “Request-Response” mechanism is a two-way communication between Module Instances. The calling Module Instance Requests an operation and the called Module Instance provides a Response. The Requesting Module Instance (sender of the Request) is named the “Client”, and the providing Module 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 and two mechanisms for Response operations (which provide synchronous and asynchronous behaviour at the Client and Server respectively). The details of these are described in the following sections. Note that the choice of mechanism for either a Request or a Response operation can be completely independent of each other.

For each Request-Response, the set of possible Clients and Servers are identified at design time, as is the type of the mechanism e.g. synchronous/asynchronous, immediate/deferred.

When a Client performs a Request, if the Server is a Module Instance within the same Component instance, then this Server is used. However, if the Request is connected to a Service, there may be multiple possible Servers available; only the Server connected with the Service Link with the lowest value of Wire Rank is used (known as the active Server, see Section 10). A Response from a Server is only sent to the particular Client that has issued the Request.

A call may fail if the Server is not available. The Client is notified of the failure of the call, and the fault reported to the fault-management Infrastructure.

The client Container instance may implement a timeout (determined by the maximum Response time defined by the required QoS) in order to unblock the Client of a Synchronous Request-Response if no Response is received.

The four types of Request-Response are detailed in the following sections.

7.3.1 Synchronous Request

In the case of a Synchronous Request, the Client Module Instance is blocked until the Response is received, as shown in Figure 5.

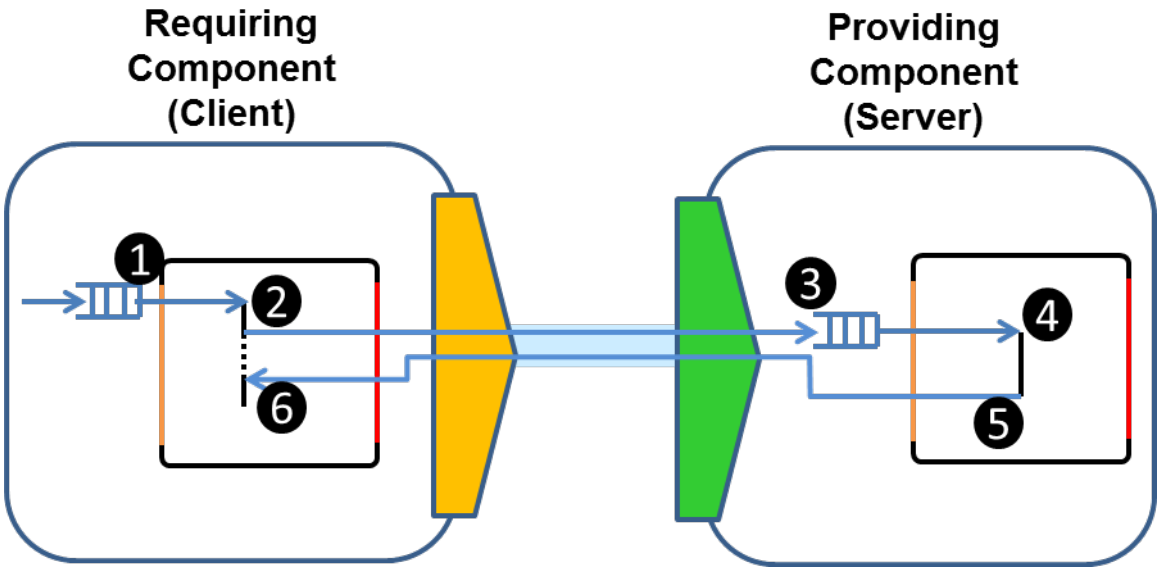

Figure 5 – Synchronous Client Request-Response

Figure 5 shows, at **point 1**, a Module Operation being invoked on the Client Module as a result of some other activity. During this execution, the Module Instance performs a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked.

The Request operation is connected via a Service Link to the Server Module Instance, whereby the Request operation is queued in the Server Module Instance Queue, at **point 3**.

The Request operation will be invoked on the Server Module Instance at **point 4**, which, in this example, will return with the Response at **point 5** (Immediate Response). Once the Response is received by the Client Module Instance, it will become unblocked and can continue its execution at **point 6**.

7.3.2 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 Module Instance, as shown in Figure 6.

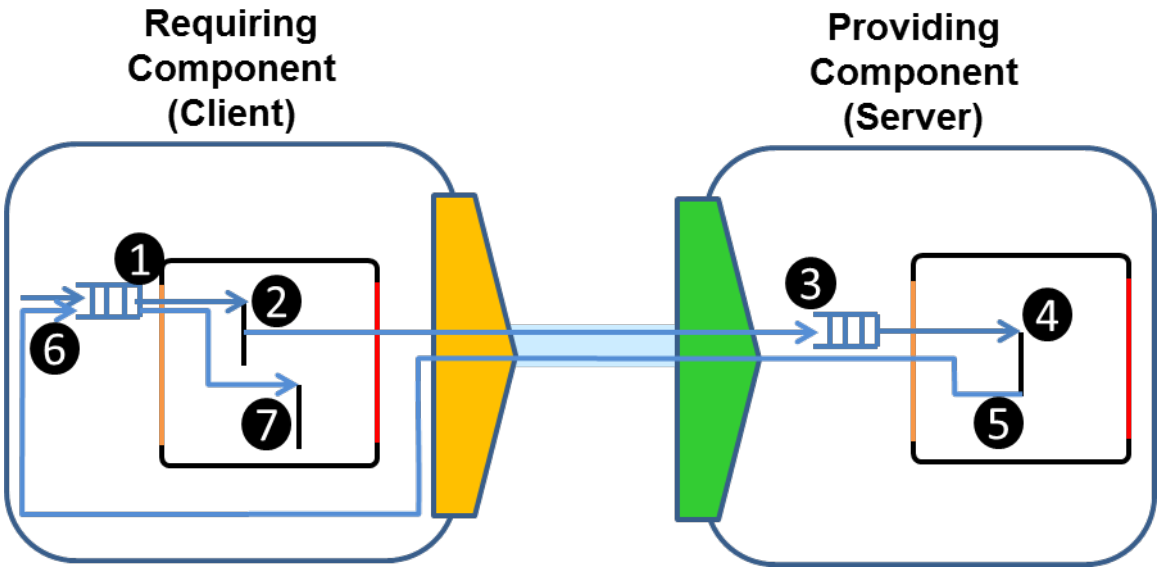

Figure 6 – Asynchronous Client Request-Response

Figure 6 shows, at **point 1**, a Module Operation being invoked on the Client Module Instance as a result of some other activity. During this execution, the Module Instance performs an Asynchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance does **NOT** block meaning it can finish its execution of the invoked operation.

The Request operation is connected via a Service Link to the Server Module Instance, whereby the Request operation is queued in the Server Module Instance Queue, at **point 3**.

The Request operation will be invoked on the Server Module Instance at **point 4**, which, in this example, will return with the Response at **point 5** (Immediate Response). The Response will then be placed in the Client Module Instance Queue at **point 6**, and the Response call-back operation will be invoked on the Client Module Instance at **point 7**.

7.3.3 Immediate Response

In the case of an Immediate Response, the Server executes the required functionality and provides an immediate Response (analogous to the Synchronous Request on the Client side; in that the server will be blocked from processing other Module Operations until a response is provided).

Note that this behaviour is shown in section 7.3.1 (with regard to the Immediate Response of the Server).

7.3.4 Deferred Response

In the case of a deferred Response: the Server may defer the provision of the Response e.g. where it needs to invoke a Request-Response Service in order to provide the Response. In this case the Server may continue to execute other functionality before providing the Response, as shown in Figure 7.

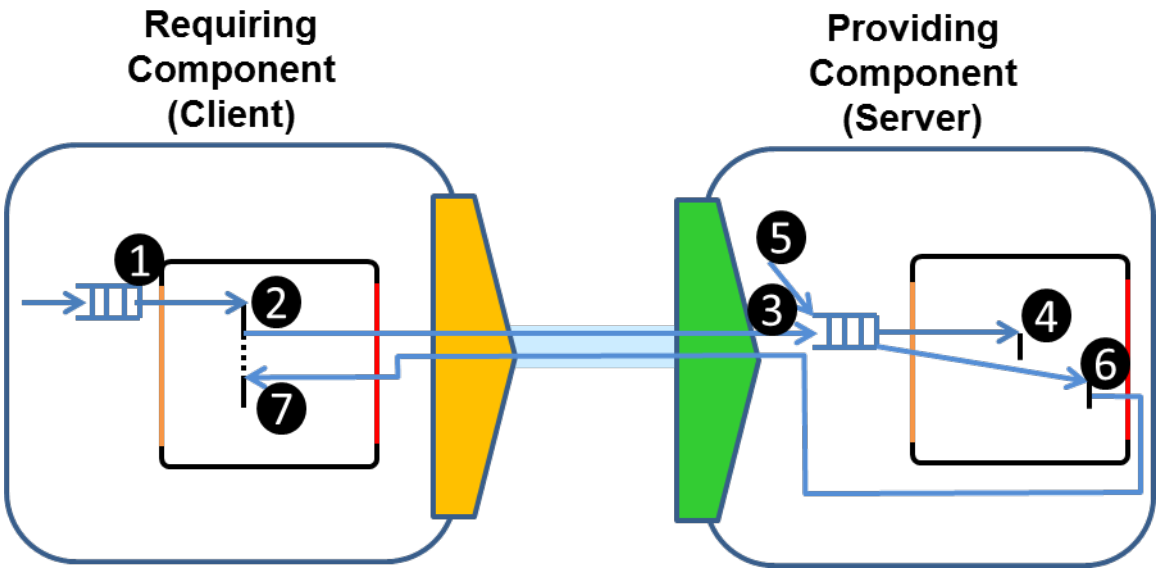

Figure 7 – Deferred Response Server Request-Response

Figure 7 shows, at **point 1**, a Module Operation being invoked on the Client Module Instance as a result of some other activity. During this execution, the Module Instance performs, in this example, a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked.

The Request operation is connected via a Service Link to the Server Module Instance, wherein the Request operation is queued in the Server Module Instance Queue, at **point 3**.

The Request operation will be invoked on the Server Module Instance at **point 4**. The Server may, by design, use a Deferred 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 Response cannot be computed until receipt of the Response occurs.

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

7.4 Versioned Data Publication

The Versioned Data publication mechanism allows Module Instances to share typed data, according to a concurrency-safe read-write paradigm. A reading Module Instance is named a “Reader”, and a writing Module Instance is named the “Writer”.

The Writer can request a local copy of the data and subsequently commit or cancel any changes made. This write action is atomic and independent of any other updates to the data-set. The Versioned Data writes are timestamped (timestamp is performed by the ECOA Software Platform).

A Reader can also request a local copy of the data, which will be the latest data value(s) at the time of the read request. This local copy will not be affected by any subsequent changes to the data-set (i.e. by a Writer updating the data-set). Note that although it is possible for the Reader to modify its local copy of the data, it is not able to update the global data-set.

Writers and Readers have to specify the beginning and the end of each (read or write) access to the data. The mechanism is “wait-free”: no Writer or Reader is blocked waiting for another Writer or Reader to release the data or waiting for the underlying synchronisation mechanism to update the local data-set.

It is possible for multiple instances of the same Versioned Data repository to exist in an ECOA system e.g. where the Readers are on different Computing Nodes. When the access begins, the Writer or the Reader always gets the latest copy of data available locally. The timestamp enables the caller to determine the freshness of the data. The Reader gets an error if data has never been received or initialized.

Note that at the moment the data is requested, there is no guarantee that it is synchronised with the latest update, particularly in a distributed system. Where there are multiple Writers the timestamps can be used by any Readers to determine the order in which the data was published.

Any copy that is made to enable a read or write access is isolated, in that it will not be changed by any concurrent modifications of the data, and local changes will not cause the Versioned Data repository to be updated. The local copy is discarded after the read or write has been completed.

A write access ends with two possible alternatives:

- “publish” – modifications made by the Writer are published to the Versioned Data repository
- “cancel” – the modified local copy of the data is discarded without making any modifications to the Versioned Data repository.

Data publications are atomic; “simultaneous” publications of the same dataset cannot corrupt the data content. This behaviour may require support from the Infrastructure.

An optional attribute (maxVersions) may be set in the Component Implementation as an attribute of the data read/write operation to specify the maximum number of concurrent read or write accesses that may occur. The default is one access per operation e.g. a Read must be released before the next Read starts. A read or write access Request may fail if a new local copy of the data cannot be created. In this case, a null Data Handle is returned, the Client is notified of the failure of the call, and the fault reported to the fault-management Infrastructure.

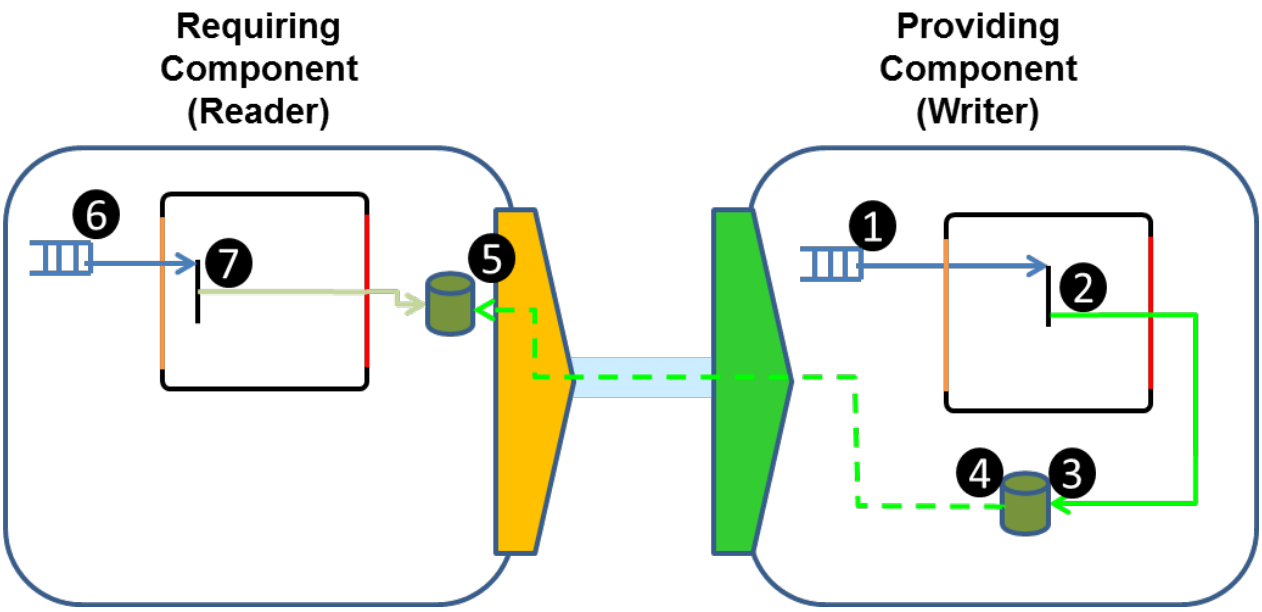

Figure 8 – Versioned Data Behaviour

Figure 8 shows, at **point 1,** a Module Operation being invoked on the Writer Module Instance as a result of some other activity. During this execution, the Module Instance performs a publish Container Operation at **point 2**. The data is written to the Component instance local copy of the repository at **point 3**. The Infrastructure is then responsible for copying the data to any requiring Components (which may not be immediate depending upon the implementation of the Infrastructure) at **point 4** and **point 5** respectively. Also note that the Infrastructure may optimise this database management such that the 'local' copies are one in the same where the components are deployed in the same protection domain.

Independently, of the Writer, the Reader Module Instance can read from the Versioned Data repository. **Point 6** shows a Module Operation being invoked on the Reader Module Instance by some means (e.g. an Event Received). During this execution, the Module Instance performs a read Container Operation at **point 7**.

7.4.1 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 Module Instance, which also contains a reference to the latest version of the data.

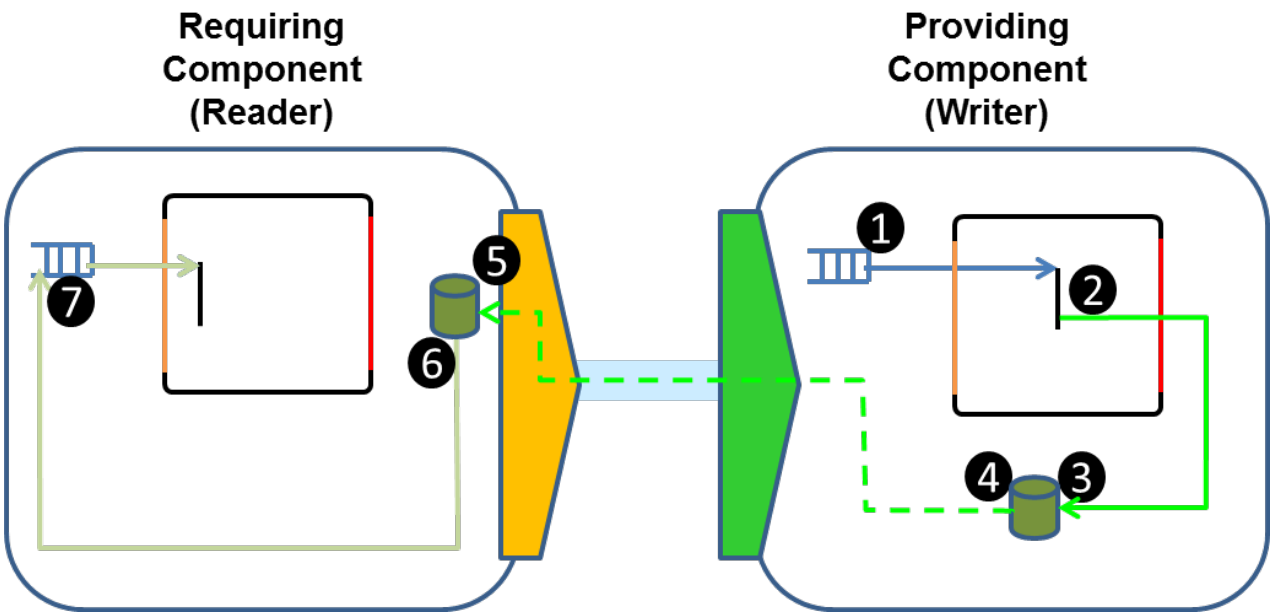

Figure 9 – Notifying Versioned Data Behaviour

Figure 9 shows, at **point 1,** a Module Operation being invoked on the Writer Module Instance as a result of some other activity. During this execution, the Module Instance performs a publish Container Operation at **point 2**. The data is written to the Component instance local copy of the repository at **point 3**. The Infrastructure is then responsible for copying the data to any requiring Components (which may not be immediate depending upon the implementation of the Infrastructure) at **point 4** and **point 5** respectively.

When the requiring Component receives the updated data (and as the Reader Module Instance has defined the operation to be notifying) a notification Event is generated at **point 6** which is queued on the Reader Module Instance Queue at **point 7**. This invokes the notification Module Operation, which also includes a copy of the updated Versioned Data.

Note that specifying a Versioned Data Read operation as notifying, does not preclude the use of standard Container Operations for getting a read-only copy of the data at any time, as detailed in section 7.4.

7.5 Trigger

Triggers generate periodic Events which can be used to invoke some functionality provided by a Module Instance or set of Module Instances. The Trigger can generate Events which are queued to Module Instances within the same Application Software Component and/or generate Events which are queued to Module Instances in different Application Software Components via a Service e.g. where a single central Trigger is used to coordinate the execution of functionality of multiple Components, in the manner of a “central clock”. The Trigger behaviour for the case of generating events within a same Component is shown in Figure 10.

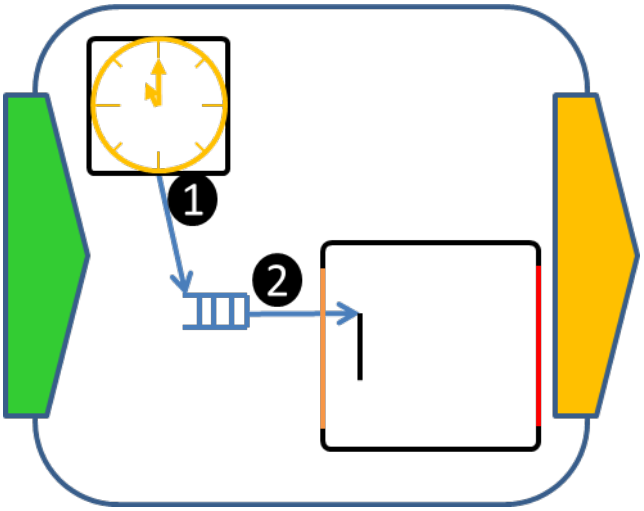

Figure 10 – Trigger Behaviour

Figure 10 shows, at **point 1**, the Trigger Instance sending an Event to the Receiver Module Instance Queue. This causes the Module Operation connected the Trigger to be invoked on the Receiver Module Instance at **point 2**. The Trigger Instance will generate the Event at a periodic interval as defined by the Component implementer.

The Trigger Instance is provided by the underlying Infrastructure to generate an Event at a set time interval. The Trigger Instance exhibits a sub-set of the Module interface, to enable the Supervision Module to control the Module Lifecycle of the Trigger.

Note that the Trigger can be connected to one or more Module Operations and/or Service Operations.

7.6 Dynamic Trigger

A Dynamic Trigger sends an Event after a given delay (known as the `out` Event) from the receipt of an input Event (known as the `in` Event). The `in` Event specifies the delay time. A Dynamic Trigger may also receive a `reset` Event, which will purge all unexpired delays.

It is possible for multiple Module Instances to:

- Send `in` and `reset` Events to the same Dynamic Trigger.
- Receive the same `out` Event.

As with the periodic Trigger, the Dynamic Trigger can generate Events which are queued to Module Instances within the same Application Software Component and/or generate Events which are queued to Module Instances in different Application Software Components via a Service.

Multiple occurrences of the same `in` Event may be queued waiting for the delays to expire. A `reset` Event is used to purge all waiting `in` Events.

The first parameter of a Dynamic Trigger is the delay. The remaining parameters can be any pre-defined type. The `out` Event is generated with exactly the same parameters as the `in` Event, except that the first (delay) parameter is omitted. The `out` Event is sent at the time resulting from adding the timestamp of the `in` Event and the delay time. The timestamp of the `out` Event is the time at which the Event is sent by the Dynamic Trigger.

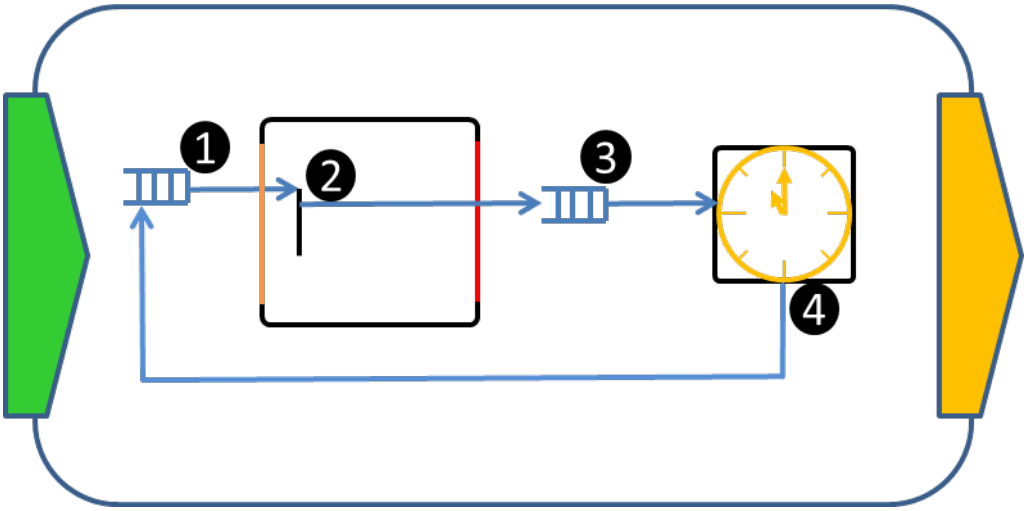

Figure 11 – Dynamic Trigger Behaviour

Figure 11 shows, at **point 1,** a Module Operation being invoked on a Module Instance as a result of some other activity. During this execution, the Module Instance performs, at **point 2**, a Container Operation to Request the Dynamic Trigger Instance to send a Trigger Event after a given period (`in` Event). The `in` Event is queued in the Dynamic Trigger Instance Queue, at **point 3**. After the delay time has expired, the Dynamic Trigger Instance will generate an `out` Event to the Receiver Module Instance Queue, shown at **point 4**.

7.6.1 Dynamic Trigger Operations

The Dynamic Trigger can be considered as a Module whose operations are:

- EventReceived in
  - On reception, the Dynamic Trigger sets the trigger with the expiration delay
  - Parameters:
    - delay : ECOA:duration (seconds, nanoseconds)
    - p1, p2, etc: ECOA types
- EventSent out
  - The Dynamic Trigger sends an “out” Event at time "timestamp(arrival of “in” Event at Dynamic Trigger level)+delay" in association with an “in” Event previously received. The timestamp of "out" is the time at which "out" is sent.
  - Parameters:
    - p1, p2, etc: are identical (number, types and position identical to those of the “in” Event)
- EventReceived reset
  - On reception, the Dynamic Trigger cancels the trigger settings for all "in" Events already previously received and not yet expired.

The transmitted Events ‘in’ and ‘out’ can have several unspecified parameters (p1, p2, etc):

- These parameters are sent as they are by the Dynamic Trigger.
- The number of parameters and their types are defined in the Component implementation model at instance definition level (see below).

From an XML point of view, the Dynamic Trigger Module definition looks like the following definition. Note that this moduleType definition is implicit and is managed directly by the Infrastructure.

```
<moduleType>
 <Operations>
  <EventReceived name="in">
   <input name="delay " type="ECOA:duration"/>
   <input name="param1" type="T1"/>
   <input name="param2" type="T2"/>
    ...
  </EventReceived>
  <EventReceived name="reset"/>
  <EventSent name="out">
   <input name="param1" type="T1"/>
   <input name="param2" type="T2"/>
    ...
  </EventSent>
 </Operations>
</moduleType>
```

7.6.2 Dynamic Trigger management

As any other Module, the Dynamic Trigger:

- Has its own lifecycle – It can receive lifecycle Events (START, STOP, etc) as specified in section 8.1.2,
- Must be supervised by the supervisor Module,
- Must be deployed by defining its Module priority and its protection domain.

7.6.3 XML definitions of Dynamic Trigger Instance and associated links

A Dynamic Trigger Instance is defined with the tag <dynamicTriggerInstance > within the Component implementation model (component.impl.xml); it is then used as any other ordinary Module.

Parameters of a Dynamic Trigger Instance are:

- Maximum number, named `size`, of waiting Events (outside possible queuing at network level)
  - By default: 1
  - Any Events that cause the maximum number of pending Events to be exceeded are discarded; an error is logged by the Infrastructure.
- Minimum and maximum values of the “delay” respectively named `delayMin` and `delayMax`
  - Statically defined at design time, to be used by early verification
  - For each received “in” Event, the Dynamic Trigger Module checks that the expiration date is compatible with these constraints (if not, an error is logged by the Infrastructure and the Event is not taken into account).

- Defined in seconds. Engineer notation is supported to ease the readability (type xsd:double).

The XML snippet below provides an example of a Dynamic Trigger Instance with one integer parameter.

```
<dynamicTriggerInstance name="delayResult"
      modulePriorityRanking="20"
      size="10" delayMin="100e-3" delayMax="200e-3">
  <parameter name="p1" type="int32"/>
</dynamicTriggerInstance>
...
<EventLink>
  <senders><moduleInstance name="producerComputer" OperationName="result"/></senders>
  <receivers><dynamicTrigger name="delayResult" OperationName="in"/></receivers>
<EventLink>
<EventLink>
  <senders><dynamicTrigger name="delayResult" OperationName="out"></senders>
  <receivers>
    <moduleInstance name="consumerComputer" OperationName="result"/>
  </receivers>
</EventLink>
<EventLink>
  <senders><moduleInstance name="managerComputer" OperationName="reset"/></senders>
  <receivers><dynamicTrigger name="delayResult" OperationName="reset"/></receivers>
<EventLink>

 ...
```

7.7 Interactions within Components

Although the majority of examples shown in the preceding sections display interactions between Module Instances in multiple Components (using Services), the exact same interactions can occur within a Component boundary (Module to Module communications).

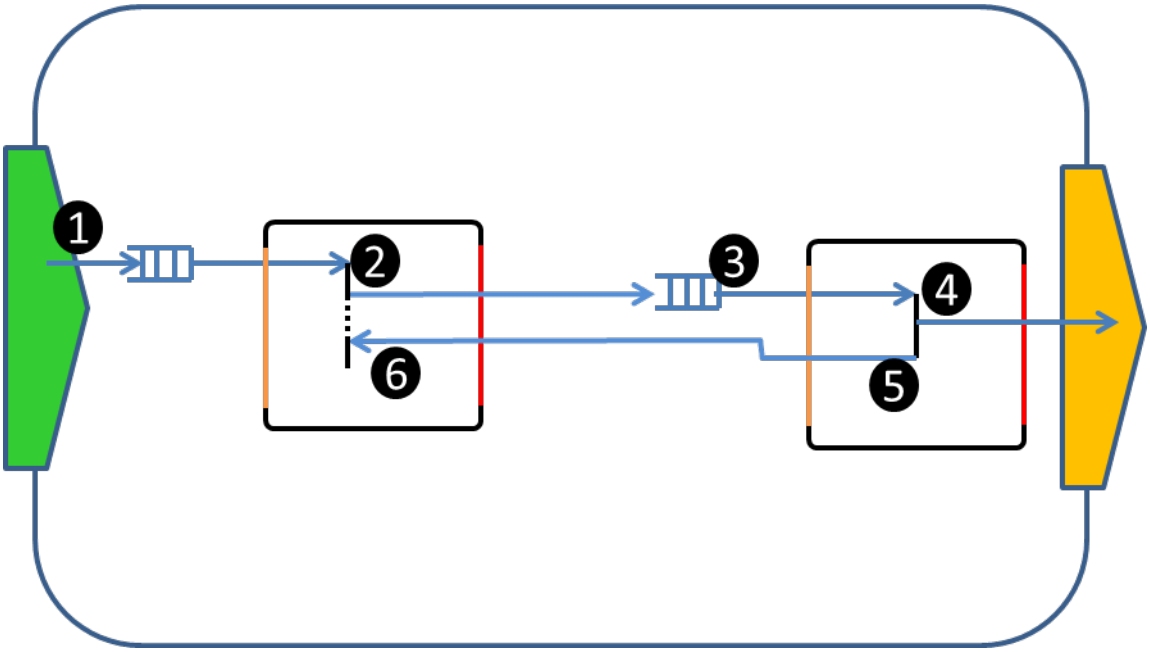

Figure 12 – Interactions within Components – Synchronous Request-Response

Figure 12 shows the interactions of a Synchronous Request – Immediate Response operation between two Module Instances within a Component. At **point 1**, a Module Operation is invoked on the Client Module Instance from a Service Operation.

During this execution, the Module Instance performs a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked.

The Request operation is connected to another Module Instance within the Component. The Request operation is queued in the Server Module Instance Queue, at **point 3**. The Request operation will be invoked on the Server Module Instance at **point 4**, which can peform any processing required in order to produce the Response.

In this example, at **point 4**, the Module Instance invokes a Container Operation to perform an Event Send. The Server Module Instance provides the Response at **point 5** (Immediate Response) when the Module Operation completes and returns control to the Container. Once the Response is received by the Client Module Instance, it will become unblocked and can continue its execution at **point 6**.+

7.8 Component and Module Properties

Components may be instantiated multiple times within a system. Component Properties provide a means to tailor the behaviour of a Component instance. A Component Property is declared as part of the Component Definition.

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

As part of a Component Implementation a Module Type can have Module Properties declared, which can then be used to tailor different behaviour for each Module Instance.

For each Module Instance, a Module Property can either reference a Component Property, or be assigned a value at Component implementation time (design time) which cannot be changed during execution. Note that in order for Component Properties to be accessed; a Module Property must be created to reference the Component Property.

Module Instances can then access Module Properties, and consequently Component Properties if referenced, via the Container Interface at run-time. The Software Interface Reference Manual (Reference 9) contains more detail regarding Properties.

8 ECOA System Management

8.1 Lifecycle

A Component instance has a lifecycle which enables a hierarchical management structure to be implemented if required. The concept of the Component-level lifecycle is described in section 8.1.1.

A Component Instance is composed of Module Instances. The Runtime Lifecycle of a Module Instance defines its runtime state. The Module Lifecycle is described in Section 8.1.2.

8.1.1 Component Runtime Lifecycle

A set of lifecycle states have been defined to facilitate the creation of reusable Application Software Components.

Each Component provides a standardized Component lifecycle Service, which offers a number of operations. This lifecycle Service can be used by application software management Components to manage the Component Lifecycle in the same way as any other Service.

The Component Lifecycle is illustrated in Figure 13.

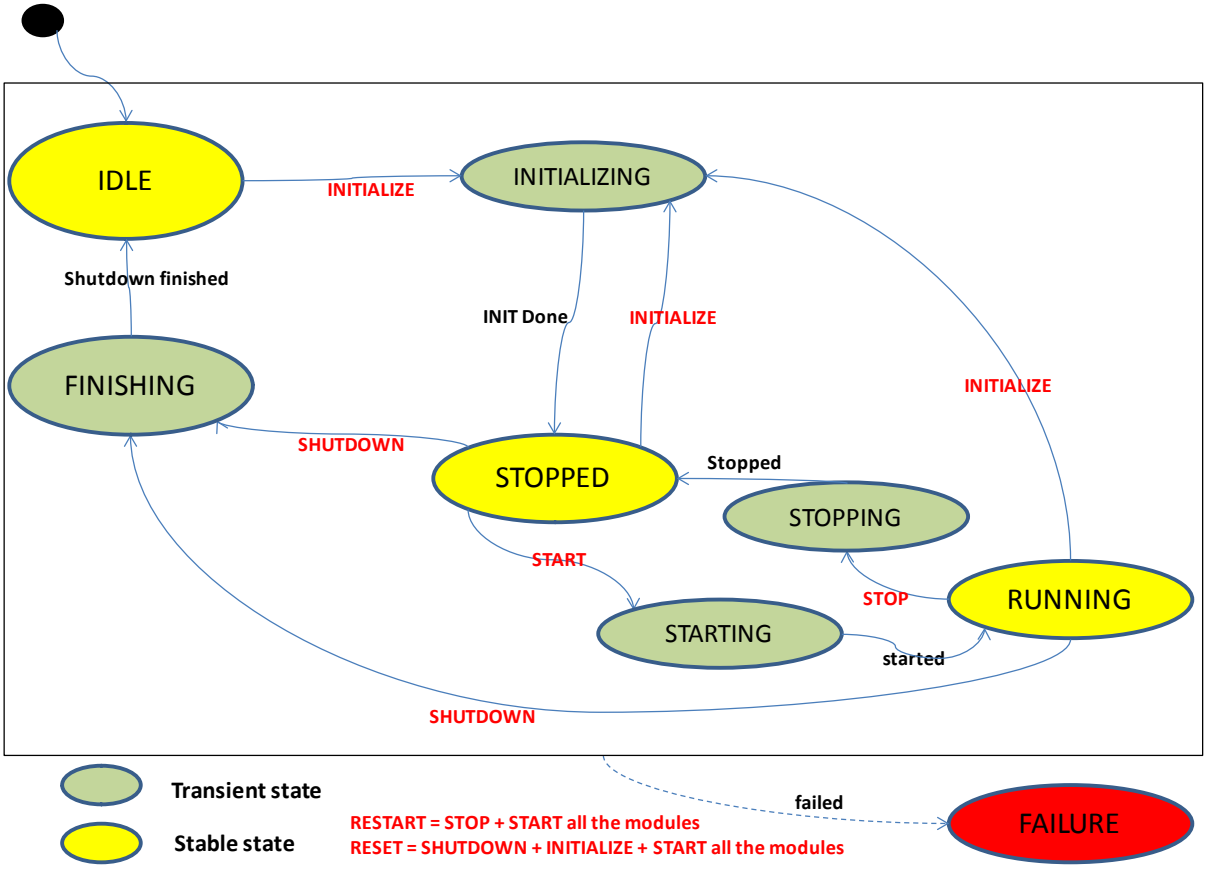

Figure 13 – Component-level lifecycle states

A Component instance has four possible stable states:

- **IDLE**
- **STOPPED**
- **RUNNING**
- **FAILURE**

A Component instance also has four possible transient states:

- **INITIALIZING**
- **STARTING**
- **STOPPING**
- **FINISHING**

A Component instance can transition between the four stable states via the transient states as shown in Figure 13. Every Component Implementation must contain a Supervision Module; a Component Implementation may comprise a single Module Instance, which by default is the Supervision Module. The states and transitions are managed by the Supervision Module. The condition by which the Component Runtime Lifecycle state is determined is dependent upon the internal design of the Component and, usually, the Module Runtime Lifecycle state of its constituent Modules.

The transient states shown in Figure 13 are where the Supervision Module is in the process of changing the state of the Component instance from one stable state to another.

Any Lifecycle Command that is not a valid transition from the current state (as shown in Figure 13) is ignored by the Supervision Module.

8.1.1.1 Component Lifecycle Service

The Component lifecycle Service is composed of:

- The current **state** of the Component instance (Versioned Data)
- A set of received Events (Received by Provider – see section 7.2.2) to command changes to the state of the Component instance:
  - **initialize_component**
  - **start_component**
  - **stop_component**
  - **restart_component**
  - **reset_component**
  - **shutdown_component**
- A set of notification Events (Event Sent by Provider – see section 7.2.1) sent on completion of a state change (new Component-level lifecycle state reached):
  - **initialized**
  - **started**
  - **stopped**
  - **idle**
  - **failed**

These Lifecycle Service Operations are linked to a set of Supervision Module Operations and Supervision Container Operations that allow the Supervision Module to manage the Component state.

In addition a single Container Operation is provided to allow a Supervision Module Instance to set the Component State. This operation checks the state transition is valid, and will publish the new Component State, followed by sending a notification Event (as required) – See Reference 9.

8.1.1.2 Component Lifecycle State Machine

The Component Lifecycle state machine implements the Component Lifecycle, and manages the states of the Module instances. The state machine is implemented by the Supervision Module within a Component, and the details of when a Component is in one of the main states, of transitions between them is specific to that Component.

For instance, a Component instance could be in the RUNNING Component state when all of its Module Instances are in the RUNNING Module state.

Alternatively, a Component instance may be designed to be in the RUNNING Component state with only a sub-set of its Module Instance in the RUNNING Module state. This may be used to implement a way of managing resources when in different modes within the Component.

For Component designs that have a simple relationship between the states of the constituent Modules and the state of the Component, the Supervision Module lifecycle operations may be quite generic and could be generated mechanically.

8.1.2 Module Runtime Lifecycle

Figure 14 illustrates the Module Runtime Lifecycle.

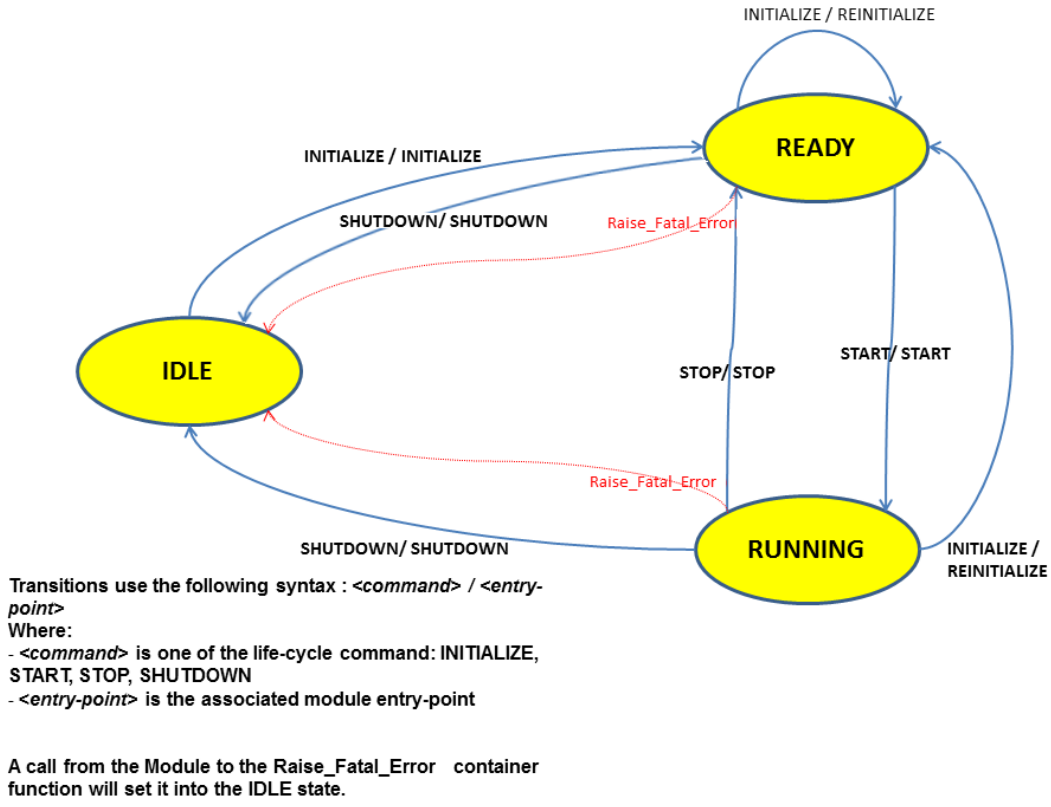

Figure 14 – Module Runtime Lifecycle

A Module Instance has three possible states:

- **IDLE**
- **READY**
- **RUNNING**

A Module Instance can transition between these states as shown in Figure 14. The states and transitions are managed by the Container, which invokes Module Interface entry points for each state change.

The Module Interface contains entry points that are invoked by the Container as a result of a Module Lifecycle state change.They are:

- **INITIALIZE**
- **REINITIALIZE**
- **START**
- **STOP**
- **SHUTDOWN**

Note: For an INITIALIZE command, the Container will invoke the INITIALIZE entry point if the Module is in the IDLE state and the REINIITIALIZE entry point if it is in the READY or RUNNING state.

The lifecycle of non-Supervision Module Instances within a Component instance are managed by the Supervision Module with the assistance of the Container:

In order to achieve this, the Supervision Module Container Interface provides the following operations (one set per non-Supervision Module Instance):

- **INITIALIZE**
- **START**
- **STOP**
- **SHUTDOWN**

The Supervision Module Instance may call combinations of the above operations in Response to a Component level lifecycle change Request, for example:

- **RESTART** could cause the Supervision Module Instance to **STOP**/**START** other Module Instances.
- **RESET** could cause the Supervision Module Instance to **SHUTDOWN/STOP**/**START** other Module Instances.

8.1.2.1 Module Startup

Upon start-up of the ECOA System, the ECOA Infrastructure is responsible for the allocation and initialization all the resources (threads, libraries, objects, etc.) needed to execute the functionality of the Module Instances and their Containers.

Following allocation and initialisation of resources each Module Instance is brought to the **IDLE** state.

8.1.2.2 Supervision Module Startup

When all Module Instances have been brought to the **IDLE** state, the Container commands all Supervision Module Instances to **INITIALIZE** and **START** by performing the following:

- the Container changes the state of the Supervision Module Instance from **IDLE** to **READY**, and calls the **INITIALIZE** entry point in the Supervision Module Interface
- when the entry point returns, the Container changes the state of the Supervision Module Instance from **READY** to **RUNNING**, and calls the **START** entry point in the Supervision Module Interface.
- Note that although the Supervision Module is able to manage non-Supervision Modules at any time; it is recommended that it only manages them in the RUNNING state (i.e. during or after the **START** entry point).

8.1.2.3 Non-Supervision Module Startup

Within a Component, the Supervision Module can Request non-Supervision Module Instances to **INITIALIZE** (or **REINITIALIZE** depending on the state of the Module Instance). The Module Instance then performs any actions required to initialize, or re-initialize, such as allocating further resources, setting its internal variables and/or reading its Properties to reach a functionally coherent and initialized internal state.

Following initialisation the Module Instance is **READY** and the Supervision Module can Request it to **START**. A this point, the Module Instance has the opportunity (via its **START** operation) 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 Module Instance enters the **RUNNING** state.

The following are the steps taken to change the state of a Module Instance:

- the Supervision Module Requests a Module Instance lifecycle state change via the Container Interface
- the Container changes the state of the Module Instance immediately prior to invoking the appropriate entry point in the non-Supervision Module Interface
- the Container then notifies the Supervision Module when the entry point of the non-Supervision Module Instance returns – indicating the state change is complete.

8.1.2.4 Module Run-time Behaviour

Lifecycle Events that do not represent a valid transition from the current state, as shown by the Module Lifecycle state diagram in Figure 14, are discarded, and the fault management Infrastructure will be notified (see section 8.3).

Lifecycle Events have no priority over other operations:

- On receipt of **STOP**, **SHUTDOWN** or **INITIALIZE** Events, operations already executing are allowed to complete and operation calls already queued will be executed.
- operations arriving when a Module Instance is entering the **RUNNING** state will be queued and executed after the **START** entry-point has returned.

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

8.1.2.5 Module Shutdown

A Module Instance may be shutdown in response to a non-recoverable error, when its **SHUTDOWN** entry point is called. This entry point should be used for the de-allocation of the resources used by the Module Instance (those previously allocated during **INITIALIZE**), after which the Module Instance enters the **IDLE** state. Information regarding Fault Management may be found in section 8.3.

8.1.3 *Lifecycle Example*

Figure 15 provides an example of how the Component and Module lifecycle can be used in a system in order to create a level of hierarchical management. Here, a "Manager" Component is shown, which can manage other Components in the system. The Manager Component can be auto-started by invoking the Component lifecycle operations for initialize Component and start Component on itself. Once running, it is free to command any managed Components through required lifecycle Services. In this example, the managed Components are auto-initializing, but wait to be commanded to start by the Manager Component. Note this is only one example and the lifecycle Service can be used in numerous ways, depending on system requirements.

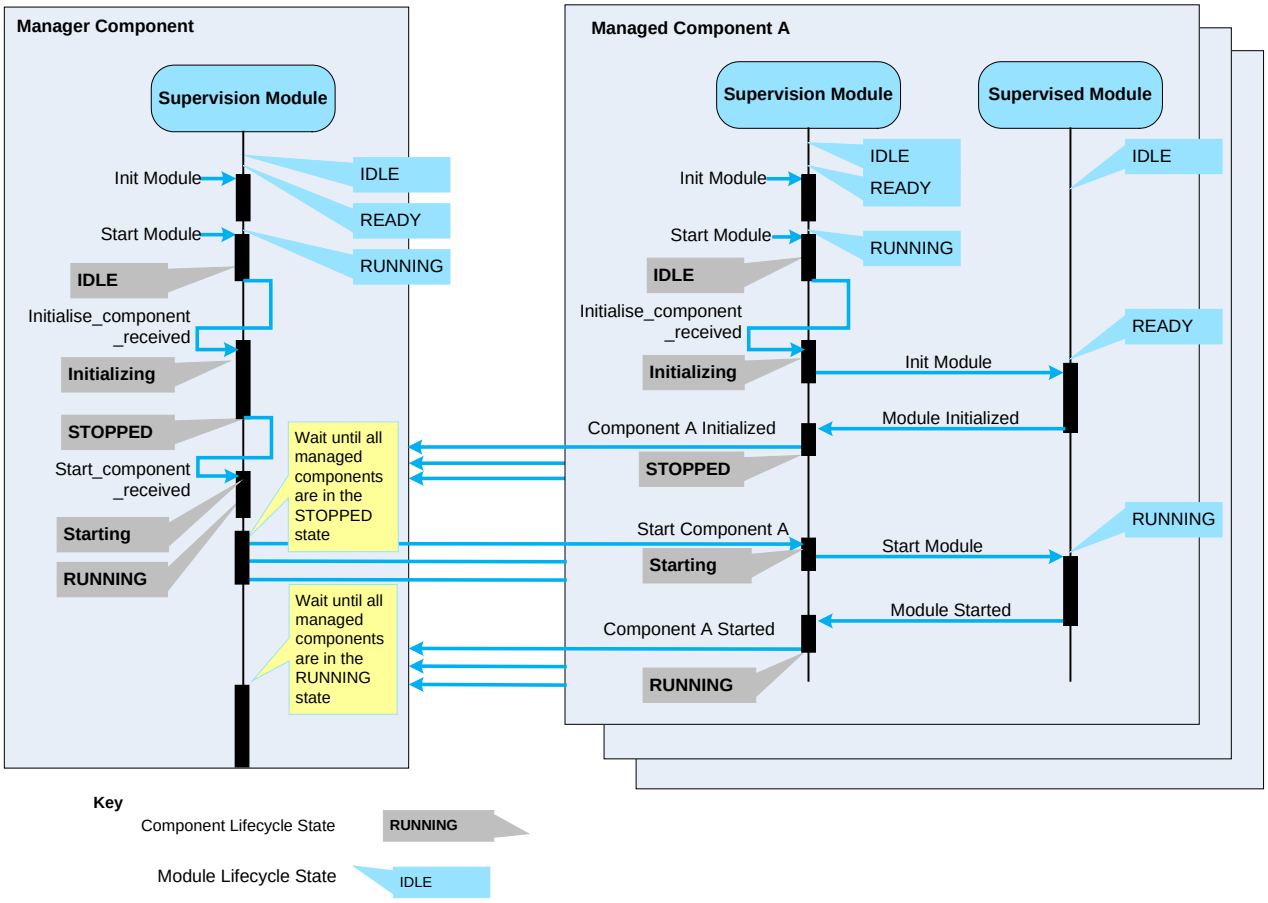

Figure 15 – Lifecycle Example

8.2 Health Monitoring

The area of health monitoring is fairly immature within the ECOA, and has currently not been discussed.

8.3 Fault Management

Management of faults is performed at various levels within the Infrastructure, the aim is to manage faults in such a way as to isolate, and minimise fault propagation between Components.

Note: the area of fault management is fairly immature within the ECOA, and the information in these sections is provisional.

8.3.1 Fault Categorization

The Infrastructure provides support to Module Instances for reporting faults. Faults can be categorised as:

- Fatal: when the Module Instance knows it cannot recover or when the Infrastructure knows it cannot recover
- Errors: where the Module may be able to recover on its own or with assistance.

8.3.2 Fault Propagation

Fault management functionality may be provided in an ECOA system:

- At the Module level by the Non-Supervision Modules
- At the Component level by the Supervision Module
- At the Protection Domain Management level by the Container.
- At the Computing Node Management level and Platform Management level by the Infrastructure.

The fault management functionality should provide recovery procedures in an attempt to handle faults at the level that they occur. Where recovery is not possible the responsibility for handling the fault is escalated to the next level. Figure 16 illustrates fault propagation in an ECOA system.

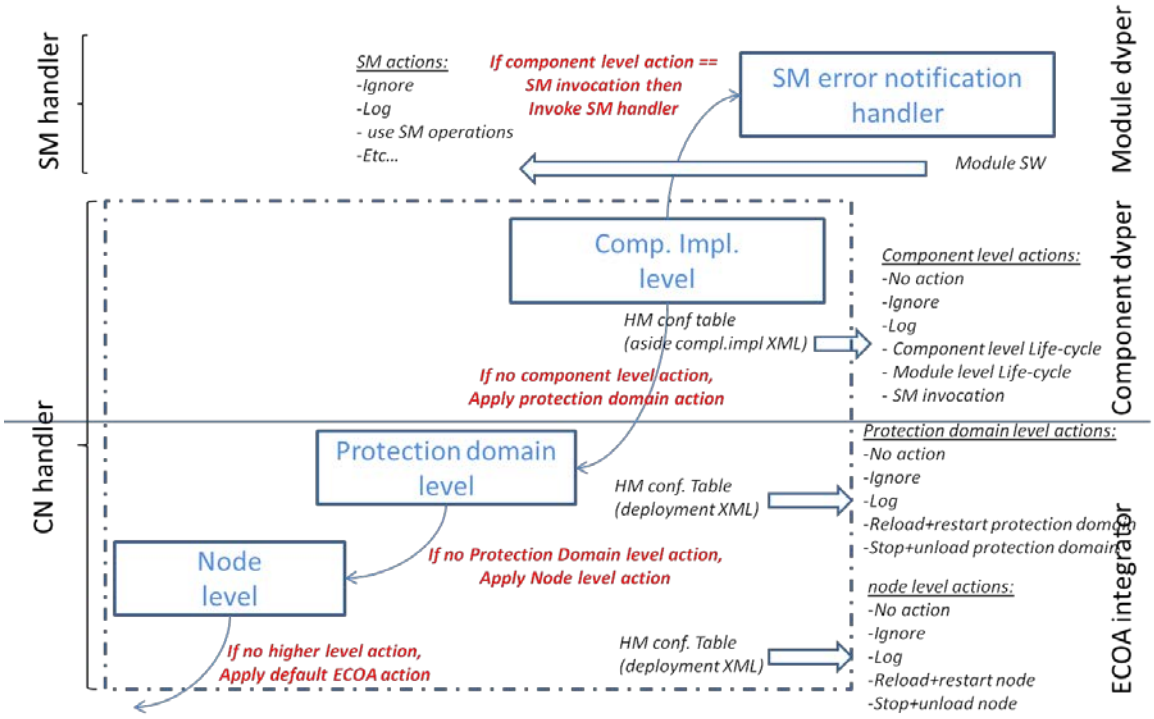

Figure 16 – Fault propagation in an ECOA System

Faults are managed in an ECOA system as follows:

- A Module Instance should be able to recover from minor errors without assistance (within the Component Implementation level). Where this is not possible the Module raises an error to the Container, and a configuration table is used to define what action the Container should take (Component level actions). The possibilities at Module level are:
    - to ignore the error

- log the error
- modify the Component State using the Component level lifecycle operations
- modify the Module level lifecycle
- raise the error to the Supervision Module
- If no action is defined at the Component Implementation level, then the fault is passed to the Protection Domain level

- The Supervision Module error notification handler is invoked by the container if the action within the Component Implementation level is to specify the Supervision Module invocation. The Supervision Module can then determine what action is to be taken (Supervision Module actions). The possibilities for the Supervision Module are to ignore the error, log the error, use Supervision Module Operations to manage the other Modules or Component lifecycle.
- At the Protection Domain level, a configuration table is used to determine what action is to be taken (Protection Domain level actions). An error detected at this level could be the result of direct detection by the Protection Domain (such as a memory violation), or be passed to it by the Component Implementation level. The possibilities at Protection Domain level are to ignore the error, log the error, reload/restart the protection domain or stop/unload the protection domain.
  If no action is defined at the Protection Domain level, then the fault is passed to the Computing Node level
- At the Computing Node level, a configuration table is used to determine what action is to be taken (node level actions). An error detected at this level could be the result of direct detection by the Computing Node Level, or be passed to it by the Protection Domain level. The possiblities at Computing Node level are to ignore the error, log the error, reload/restart the Computing Node, or stop/unload the node.
  If no action is defined at the Computing Node level, then the fault is passed to the default ECOA handler

8.4 Run-time Configuration Management

Configuration management within an ECOA system is the responsibility of the underlying Infrastructure. The configuration of the system is managed in accordance with the system management policies.

It is envisaged that a set of pre-defined alternative configurations of Service Links is available to support different functionality. The different configurations will be described by different Assembly Schemas along with system specific information regarding the scheduling of Modules.

Note: the area of configuration management is fairly immature within the ECOA, and the information in these sections is provisional.

8.4.1 Initialisation

When an ECOA system is initialized all of the Application software Components are initialized and put into their initial state (see Section 8.1 for a description of the Lifecycle). The Components then set their Services available as they become functional, which may depend on the availability of required Services.

8.4.2 Reconfiguration

Reconfiguration in an ECOA system may occur because of:

- Change of Mode: The system may reconfigure active Components by changing the Assembly Schema being used. This will result in a change to the Components and Services available, along with the configuration of Service Links between them. This is a more fundamental change of the Assembly Schema, and would only be used for gross changes in configuration.
- Loss of Service: if a Service becomes unavailable, and an alternative Provider is available as defined in the Assembly Schema, then switching between Providers will be achieved by switching between the Service Links. This reconfiguration does not require changing to an alternative pre-defined Assembly Schema; it represents the dynamic aspects of ECOA Services.

It is envisaged that Service switching is available on any ECOA conformant computing platform. However, the change of Assembly Schema (replacement of one Assembly Schema by another Assembly Schema through unload/load actions) is highly dependent upon underlying mechanisms. The way a reconfiguration is commanded is also dependent upon the underlying mechanims; it is expected that the command is sent by a management Component through a specific API or a system management Service provided by the ECOA Software Platform.

See Section 10 for further information regarding Service availability.

9 Scheduling

Support for scheduling of Modules, which are single-threaded, is provided by the underlying operating system and the ECOA Module Application Code is agnostic to the scheduling policy used. A Module Deadline is specified for each Module Instance to assist the integrator with scheduling of the Modules.

The Developers Guide (Reference 2) contains more information regarding Scheduling.

Note: the area of scheduling is fairly immature within the ECOA, and the information in these sections is provisional.

9.1 Module Deadline

Each Module Instance has a Module Deadline attribute that is used to guide a system integrator with regard to scheduling of Modules. The deadline values are expressed in time units (milliseconds), and are a measure of the time by which any Module Operation invoked on the Module is required to have completed by. This information may be used by the system integrator to determine the scheduling parameters and policies used to schedule all Modules within a deployment.

9.2 Scheduling Policy

Scheduling is the responsibility of the Infrastructure and any scheduling policy supported by the OS/Middleware may be used as required by the system integrator. Scheduling analysis of the proposed system should be carried out in the same way as normal at design time. Scheduling analysis is outside the scope of the ECOA; although it is anticipated that it would be carried out following existing, established methods. The schedulability analysis required will be dependent upon the chosen scheduling policy.

9.3 Activating and non-Activating Module Operations

By default, Module Operations are activating; the arrival of a new operation implies the execution of the associated entry-point as soon as the Module 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 Module Operation arrives to the same Module Instance through another Module Operation Link, all pending Module Operations are then processed in FIFO order and executed as any other Module 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 Module 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 `activating` on DataLink is only useful when associated to a notifying Versioned Data (attribute `notifying` set to `true`) (see §7.4.1).

10 Service Availability

The availability of Services provided by a Component instance can be set, on a per Service basis, by the Supervision Module Instance via the Container Interface. The availability of provided Services is likely to be dependent upon a combination of the Component lifecycle state and the availability of any required Services necessary in order to successfully provide its Service.

The availability of Services may be affected by run-time errors that could cause Module Instances to be shutdown.

10.1 Initialisation

During initialisation the ECOA Software Platform sets all Services as unavailable, and will propagate the availability of Services as Component instances are initialized and started. The set of Services in the system are defined by the Assembly Schema (see section 10.2), which may or may not be available at any given time.

10.2 Assembly Schema

The possible connections between the provided and required Services of Application Software Components in an ECOA system are determined by the Assembly Schema. This provides details of the Service Links (called Wires in the Assembly Schema) between Services. Service discovery may be:

- Static: where there is a single Provider of a Service. The links between providers and requirers of such Services are statically pre-determined by the Assembly Schema and are established at system start up.
- Dynamic: where there is more than one Provider of the Service and which is the active Provider is determined at run-time, when the Service Request is made. All of the possible connections are statically defined in the Assembly Schema.

An ECOA system may contain a mix of static and dynamic connections between its Services.

10.2.1 Service Links and Ranks

The links between Services are described by the Wires in the Final Assembly Schema. Each Wire has a single Requirer of a Service (identified as a source in the schema) a single Provider of the Service (identified as a target in the schema), and a Rank.

There may be multiple Providers and Requirers of the same Service; the connections between them are determined by the Wires in the Assembly Schema. Where multiple Providers (targets) are connected to the same Requirer (source) in the case of Versioned Data or Request Response, each Wire that is connected must have a unique Rank, which is used to choose a single Provider. The Provider with an available service which is connected by a Wire with the lowest Rank value is chosen in preference to the others (i.e. Active Provider). The behaviour of Events is dependent upon whether the Service Link is specified to multicast Events. If multicast is not enabled, the Active Provider concept is used.

Further detail on Rank and Service Link behaviour can be found in section 11.

10.3 Dynamic Service Availability

The Active Provider for Events, Versioned Data and Request-Response may be decided dynamically at run-time from the possible connections defined in the Final Assembly Schema. Where the current Provider of a Service (the available Provider with the lowest Rank value) becomes unavailable the ECOA Software Platform will arrange for the Provider connected by a Wire with the next lowest Rank value (if the Service is available) to be chosen as:

- the sender in the case of an Event Sent By Provider (unless multicast is enabled),
- the receiver in the case of an Event Received By Provider (unless multicast is enabled),
- the responder in the case of a Request-Response,
- the writer in the case of Versioned Data

Any changes to the Provider of a Service is notified to the Requirer. If there is no available Provider the Requirer will be notified that the Service is no longer available.

The Quality-of-Service provided by any Provider is monitored at run-time to ensure it is within the required QoS of the Requirer. If this is not the case, then a fault may be generated and reported to the Fault Management. In addition an alternative Provider may be used if one is available.

Note: the area of Quality-of-Service is fairly immature within the ECOA, and this information is provisional.

11 Service Link Behaviour

11.1 Introduction

A Service Link connects Application Software Components together. Each Service Link connects one Provided Service to one Required Service which refers to the same Service Definition (which consists of operations i.e. Events, Request Response and Versioned Data). This is shown in Figure 17.

Figure 17 – Service Links

It is necessary to specify the behaviour of each operation across the Service Links. This is because it is different for each type of operation.

11.2 Active Provider Component

The term *Active Provider* is introduced to describe the Application Software Component selected by the Infrastructure according to the following policy: it's provided Service is set as available and its Service Link has the lowest value of Wire Rank. (The value of the Rank attribute can be computed according to the deployment, for example to reflect a notion of "bonding" between Requirer and Provider). Rank is expressed as a positive integer value, whereby a low integer value represents a high ranking Service Link.

11.3 Summary of Behaviour

When a Service Definition includes an Event Sent By Provider operation, the Event (and its associated typed data) sent by any Provider is received by all Requirers linked to the Provider (the Event data is distributed from the Provider to many Requirers). If there are multiple providers, the requirer only receives Events from the Active Provider, unless the Service Link is specified to multicast Events.

Similarly, when a Service Definition includes an Event Received By Provider operation, the associated typed data sent by any Requirer is received by the Active Provider, unless the Service Link is specified to multicast Events.

When a Service Definition includes a Versioned Data operation, each providing Application Software Component that is linked to the Provided Service may supply that data. Application Software Components that are linked to the Required Service read the most recent value of the data provided by the Active Provider. When the Active Service Provider is changed to another one, Readers read the instance of the new Provider. The Containers hide the switch between instances.

For a Request-Response operation referenced in a Service Link, the Request from the Client is addressed (directed) to the Active Provider (Server). In other words, in the case of multiple eligible Providers, the Infrastructure will select one of the Providers to provide a Response.

These behaviours are summarised in Table 1.

Operation		Provider	Requirer
Event Operations	*sent_by_provider*	The Event and its associated typed data is sent to all Requirers	Receives the Event and its associated typed data from the Active Provider selected from set of eligible Providers (Servers) as determined by the Infrastructure according to the Rank attribute[1].
	received_by_provider	Receives the Event and its associated typed data from all Requirers if the Provider is the Active Provider[2].	The Event and its associated typed data is sent to all Providers
Request-Response Operations		Receives Requests from all Requirers (Clients)	Active Provider selected from set of eligible Providers (Servers) as determined by the Infrastructure according to the Rank attribute.
Versioned Data Operations		Updates data for all Requirers (Readers)	Active Provider (Writer) selected by Infrastructure according to the Rank attribute[3].

Table 1 – Behaviour across a Service Link

11.4 Examples

[1] This behavior is the default one when the flag AllEventsMulticasted is set to False (default value). When this flag is set to True, Requirers receive all events sent by the Providers for which the flag is set to True.

[2] This behavior is the default one when the flag AllEventsMulticasted is set to False (default value). When this flag is set to True, all Providers receive all events sent by the Requirers for which the flag is set to True.

[3] Providers maintain their own instance of the data : when a Provider accesses the data, it gets the value it wrote previously.

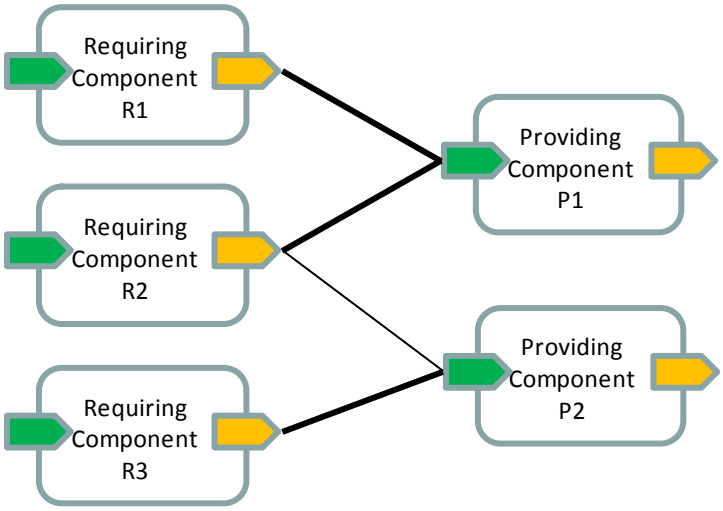

Figure 18 – Example Assembly Schema

In the above Assembly Schema, three Application Software Components R1, R2 and R3 are connected, via Service Links, to two Application Software Components P1 and P2. P1 is considered as the Active Service Provider for R1 and R2, while P2 is considered as the Active Service Provider for R3.

"Sent by Provider" Events

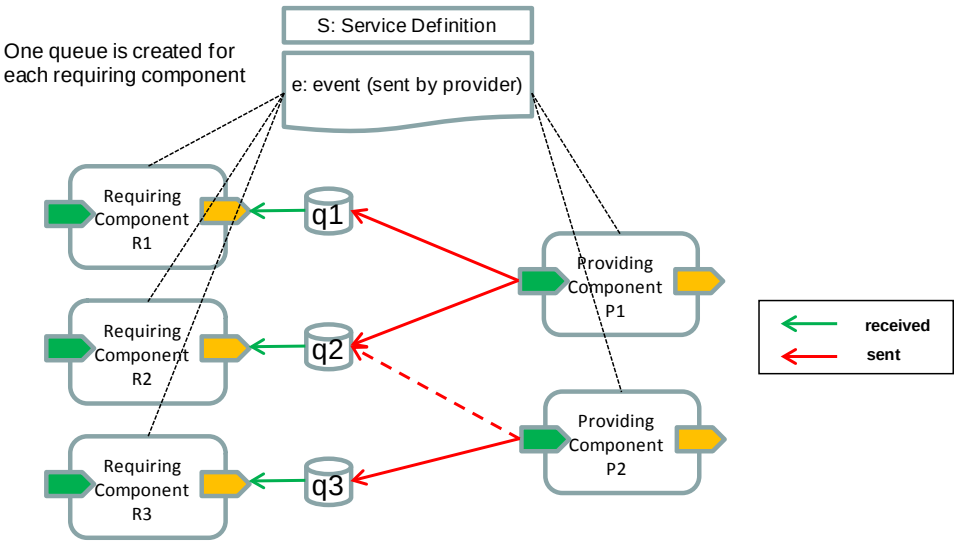

Figure 19 – Generation of an Event

An Event e “Sent by Provider” in Service S translates to:

- An Event queue (q1,q2,q3) is created for each Requiring Component (R1, R2, R3);
- An Event sent by an Active Service Provider (P1 or P2) is received by all its Requirers (R1 and R2 for P1, R3 for P2).
- An Event sent by a non-active Provider is discarded by the Infrastructure except if the *allEventsMulticasted* flag is set to true. Therefore, an Event sent by P2 is only received by R2 if the *allEventsMulticasted* flag is set to true, otherwise, it does not reach R2.

"Received by Provider" Events

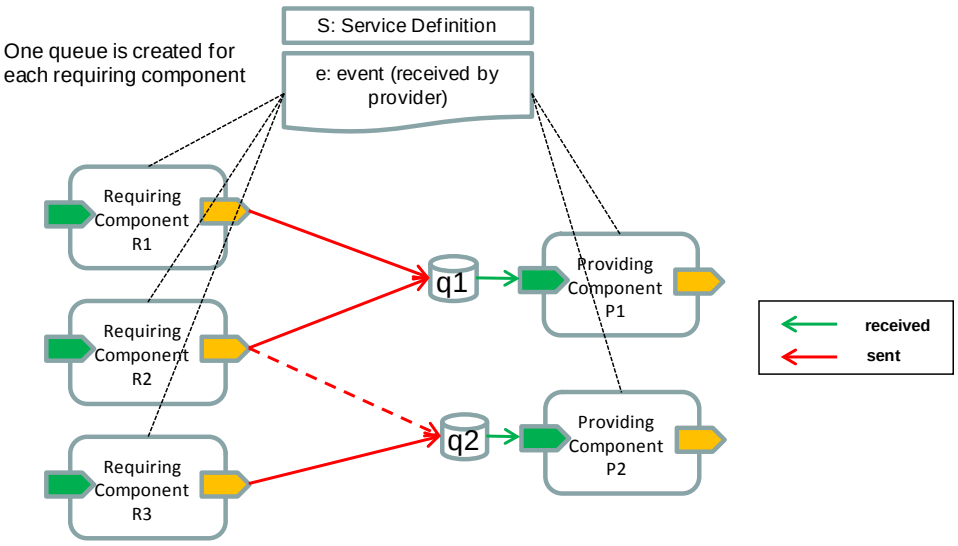

Figure 20 – Consumption of an Event

An Event “received by Provider” in a Service translates to:

- An Event queue (q1,q2) is created for each providing Component (P1 and P2);
- An Event sent by a Requirer (R1, R2, R3) is received by the Active Service Provider (P1 for R1, P1 for R2, P2 for R3)
- An Event sent by a Requirer is received by its non-active Providers if the *allEventsMulticasted* flag is set to true (P2 for R2). Otherwise, the Event does not reach non-active Providers.
- All Requirers are allowed to send the Event.

Request-Response

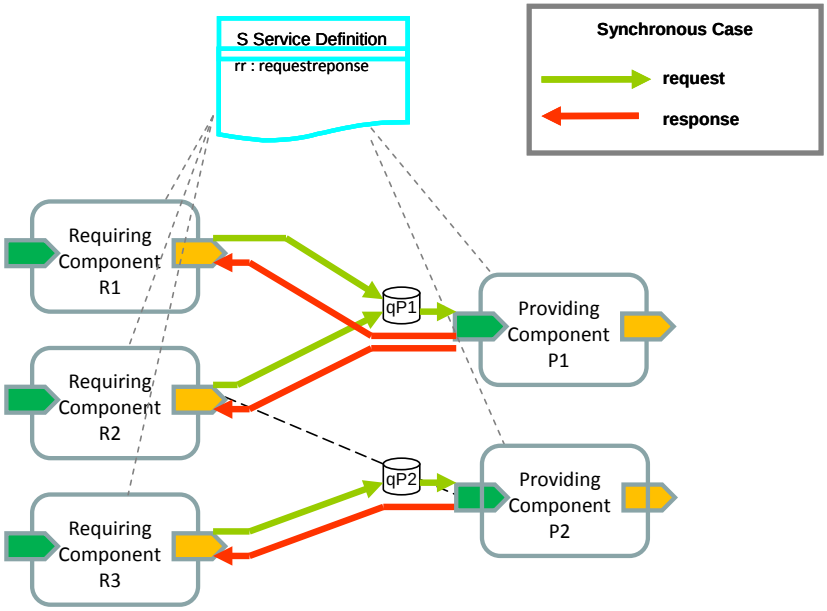

Figure 21 – Synchronous Request-Response Operation

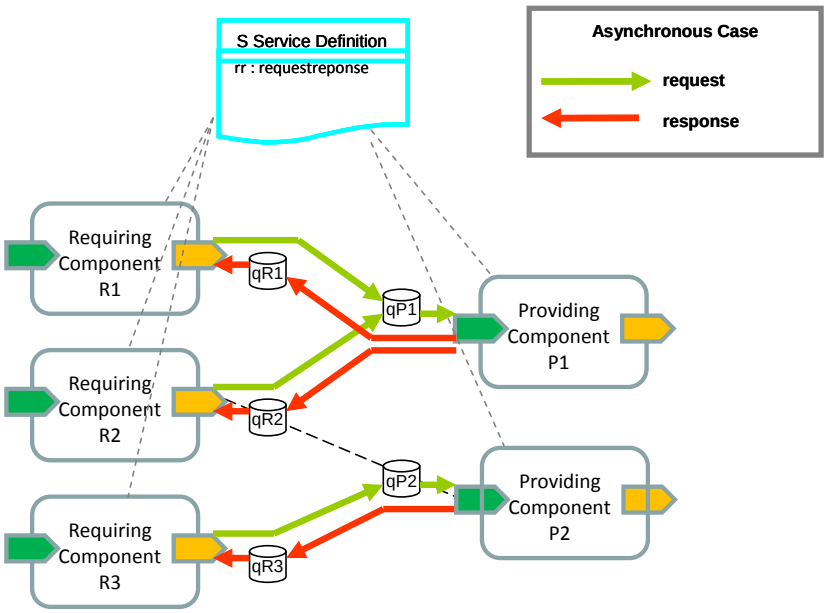

Figure 22 – Asynchronous Request-Response Operation

In Figure 21 and Figure 22, Service definition S defines a single Request-Response operation (rr). This translates to:

- Service Requirers (R1, R2 and R3) issue a Request to their current Active Provider (P1 for R1 and R2, P2 for R3).
- Based on the Rank attribute of the Service Links, the Infrastructure will determine the Active Provider for Component R2. In this example the Active Provider for R2 is P1.
- The Service Providers (P1, P2) respond to the received Requests.

- For *Synchronous* Request-Response operations, the Requiring Component is blocked until the Response is received.
- For *Asynchronous* Request-Response operations, the Requiring Component is not blocked. The Response is received at some later time and processed by a callback function.
- Requests into a Provider are queued (qP1 and qP2) until the relevant Request_Received API callback function is called by the Provider Component's Container. This will cause additional blocking delays to Requirers of *Synchronous* Request-Response operations.
- Responses to *Asynchronous* Request-Response operations are queued in the Requiring Component's queue (qR1, qR2, qR3) until the relevant Response_Received API callback function is called by the Requirer Component's Container.
- Responses to *Synchronous* Request-Response operations are not queued in the Requiring Component which will be blocked waiting in the Request_Sync API function for the Response.

Versioned Data

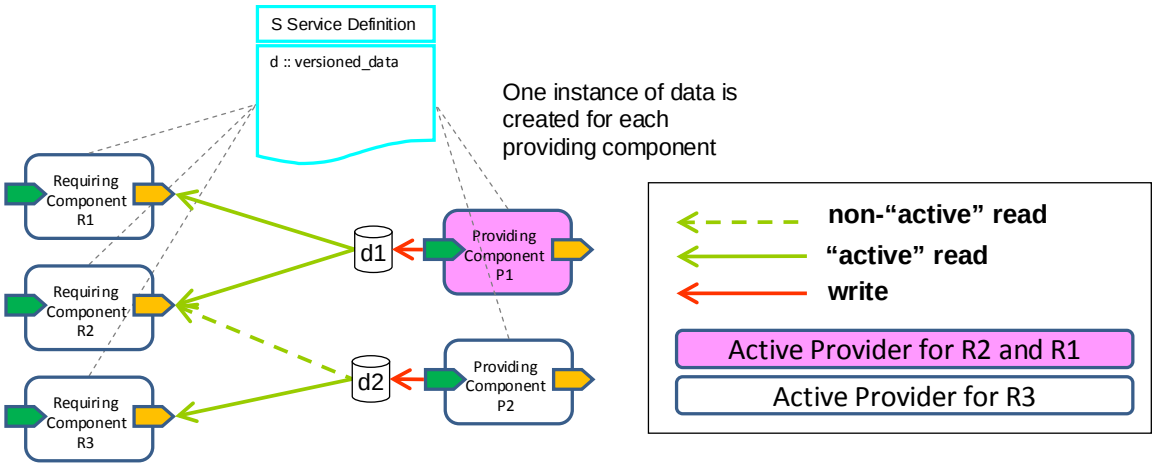

Figure 23 – Selection of Versioned Data

In Figure 23 Service definition S defines one Versioned Data operation (d). This translates to:

- Two instances of data[4] (d1 and d2) created – one per providing Components (P1 and P2)
- Based on the Rank attribute of the Service Links, the Infrastructure will determine the Active Provider for the data Reader Component R2. In this example the Active Provider for R2 is P1 (hence R2 accesses data instance d1).
- Each Versioned Data instance is written by only one Application Software Component and can be read by many Application Software Components. In this example, even if P1 is the active Provider for R2, P2 only accesses data instance d2. That means there is no underlying synchronisation between d1 and d2.

[4] This is a logical view from the point-of-view of the Component. In an actual platform implementation, the data may be physically distributed and synchronized across the processing nodes in different ways.

12 Module Operation Link Behaviour

This section shows the relationships between Service Operations and Module Operations.

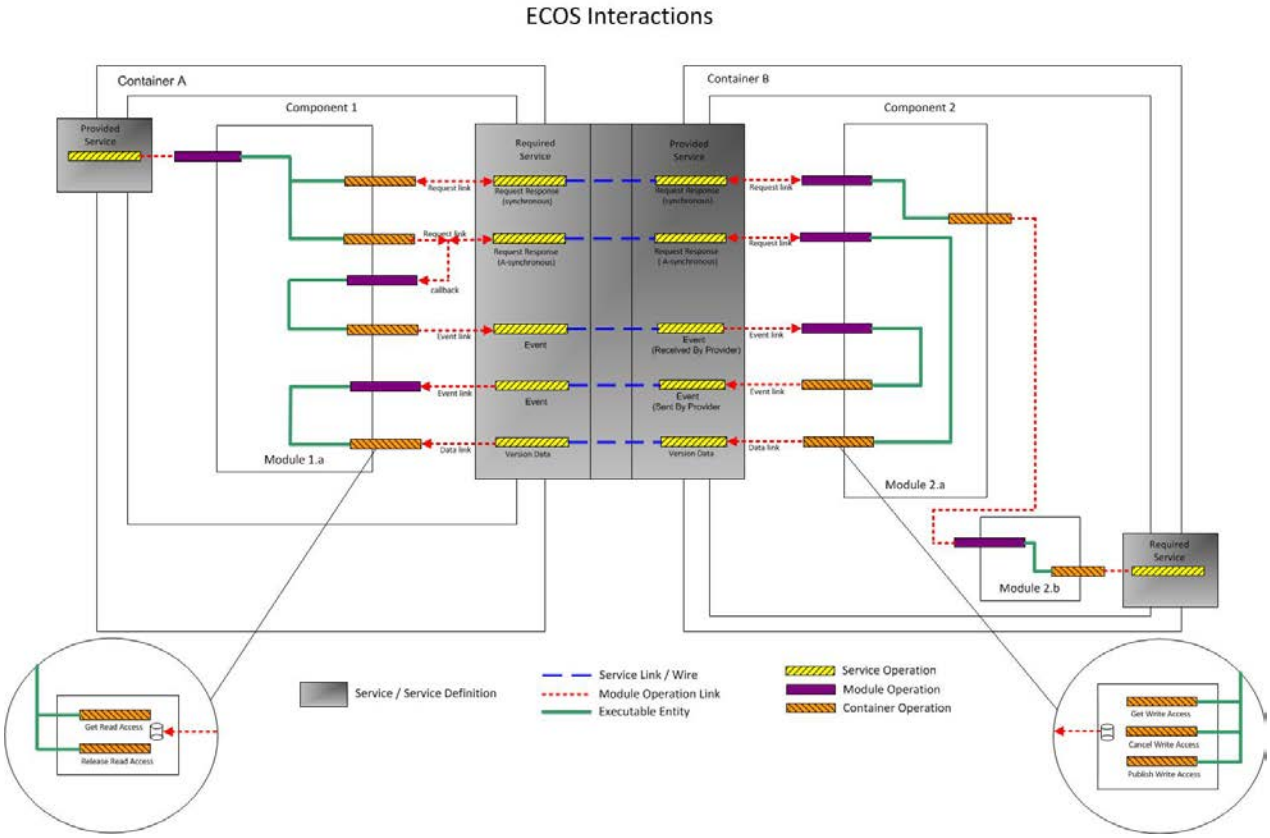

Figure 24 – Interactions between Service Operations and Module Operations

Figure 24 shows possible interactions between Service Operations and Module Operations. The following give more details on each of these:

- The Grey boxes are the Services which are collections of Service Operations and are described by Service Definitions.
- The Yellow boxes are Service Operations which are connected together using Service Links / Wires which are in Blue.
- The Purple boxes are Module Operations which are entries onto Executable Entities which are in Green.
- The Orange boxes are Container Operations which are called from an entry point of a Module Instance.
- Incoming Service Operations are connected to Module Operations using Module Operation Links.
- Container Operations are connected to outgoing Service Operations using Module Operation Links.
- Container Operations are connected to Module Operations using Module Operation Links.
- Service Operations cannot be mapped to other Service Operations using Module Operation Links.

- All Module Operation Links require Container code.
- The names of Service Operations and Module Operations which are connected together don't have to match.
- A Request Response Service Operation can only be mapped to a single Module Operation.
- Multiple Event Service Operations can be mapped to the same Module Operation.
- One Container Operation can be mapped to multiple Event Service Operations.
- Versioned Data is always Read or Written by an entry point of a Module Instance using a Container Operation.

13 Utilities

An ECOA Software Platform provides utility functions for acquiring time and for generating logs. The Software Interface Reference Manual (Reference 9) contains more detail regarding these functions.

One of the ECOA Software Platform provided functions is a method for allowing access to global time. It is a system specific decision how this global time is synchronised, and at what precision, however the ECOA assumes that time values acquired through these functions are synchronised across the system.

14 Inter Platform Interactions

In order to provide interoperability between ECOA Software Platforms, a message protocol has been defined. This message protocol requires an underlying transport protocol for its implementation. The choice of the underlying transport protocol is left to the system designer depending on system-level requirements (performance, security, etc.). As an example, a binding to the UDP transport layer has been defined. These can be found in the ELI and Transport Bindings Reference (Reference 6).

15 Composites

Note: the area of Composites is fairly immature within ECOA, and the information in these sections is provisional. Further detail of the Composite concept can be found in Reference 12, which is included for information only, as it can help provide an indication of what a Composite may encompass with regard to the ECOA.

Within the ECOA it is envisaged that a system may be constructed using many Components. In order to help with design abstractions the concept of a Composite is introduced.

A composite is described by its definition, the list of its Application Software Components and the associated Assembly Schema of these Application Software Components. The Composite will provide several Services, each one linked to one or several Services or provided by its Application Software Components. This kind of Service Link is called a Promotion Link. The Composite will require several Services required by internal Application Software Components. The link used here is also called a promotion link. An Application Software Component external to the Composite is only connected to Services provided or required at Composite level and has no knowledge of the internal Application Software Components.

Figure 25 shows an example Composite constructed with four Components.

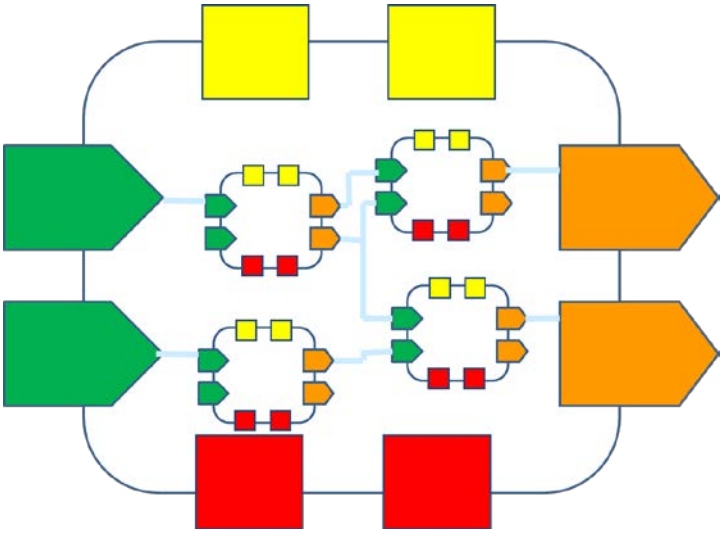

Figure 25 – A Composite

16 References

Ref.	Document Number	Version	Title
1.	IAWG-ECOA-TR-001	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume I Key Concepts
2.	IAWG-ECOA-TR-002	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume II Developers Guide
3.	IAWG-ECOA-TR-003	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 1: Ada Binding Reference Manual
4.	IAWG-ECOA-TR-004	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 2: C Binding Reference Manual
5.	IAWG-ECOA-TR-005	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 3: C++ Binding Reference Manual
6.	IAWG-ECOA-TR-006	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 4: ELI and Transport Binding Reference Manual
7.	IAWG-ECOA-TR-008	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 6: Platform Requirements Reference Manual
8.	IAWG-ECOA-TR-009	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 7: Approach to Safety and Security Reference Manual
9.	IAWG-ECOA-TR-010	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 8: Software Interface Reference Manual
10.	IAWG-ECOA-TR-011	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 9: Metamodel and XSD Schemas Reference Manual
11.	IAWG-ECOA-TR-012	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume IV Common Terminology

Table 2 – Table of ECOA references

Ref.	Document Number	Version	Title
12.	sca-assembly-1.1-spec-cd03	1.1	Service Component Architecture Assembly Model Specification Version 1.1 (available at: http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd03.pdf)

Table 3 – Table of External References