European Component Oriented Architecture (ECOA®) Collaboration Programme: Guidance Document: System Management

Date: 27/11/2017

Prepared by
BAE Systems (Operations) Limited

Contents

Figures

Tables

0 Executive Summary

System Management covers many areas of any system, from controlling the mode it is in (Initialization, Normal Operation, Shutdown, Failure etc.), through to its behaviour in response to faults.

This document covers various aspects of System Management within a system implemented using ECOA.

It is not in any way a “normative” part of ECOA, or even a definitive solution. The discussions here are purely examples of how System Management may be implemented and used in ECOA.

1 Scope

This document is intended to provide guidance for System Designers in developing an application architecture using ECOA that allows for various aspects of System Management.

The document is structured as follows:

Section 2 gives a brief introduction to the topic.

Section 3 expands abbreviations used in this report.

Section 4 provides definitions for the key terms used in this report.

Section 5 lists key documents referenced by this report.

Section 6 shows how Components can be managed.

Section 7 provides information on how Fault Handling may be implemented.

2 Introduction

This document provides some guidance and examples of what may be achieved by using standard ECOA services and mechanisms to manage a system and how it behaves in various scenarios.

3 Abbreviations

API	Application Programming Interface
DGA	Direction Générale de l'Armement
Dstl	Defence Science and Technology Laboratory
ECOA	European Component Oriented Architecture
MOD	Ministry of Defence
XML	eXtensible Markup Language

4 Definitions

For the purpose of this document, the definitions given in the ECOA Architecture Specification *(ref. [AS])* Part 2 and those given below apply.

Term	***Definition***
(currently none)	

5 References

AS	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) “ECOA” is a registered mark.

6 Component Management

6.1 Component Management Concept

ECOA does not define or mandate a way of managing the ‘mode’ or ‘state’ of a set of Components within a system.

This is seen as an advantage, in that a System Designer may choose to have no Component level state management and only use functional interactions to determine the state of a provided service. This leads to a self-managing system based upon functional state of the Service.

Alternatively a System Designer may choose to have some level of managed control over the ‘state’ or functionality of Components within the system. This could be in the form of a single level management structure, where one ‘manager’ component manages all others in the system. Or it could be a hierarchical structure where multiple levels of ‘manager’ components exist in a hierarchy.

As managed Components have a concept of their ‘state’ or ‘mode’ it is possible for them to exhibit different functional behaviour when in these different ‘states’. In addition the functional availability of any Services they provide may be dependent on this ‘state’ or ‘mode’ as well as any other required Services. This allows the System Designer to build in a finer level of control to the system such that very specific behaviours may be created.

It is also possible to have a mixture of managed and unmanaged Components within the system for areas that are more autonomous in their behaviour. It is ultimately the responsibility of the System Designer to define the best management philosophy for the system they are designing.

6.2 Component Lifecycle Service Concept

In order for Components to manage and be managed some form of interaction between them is required, which can be achieved by defining a Component Lifecycle Service. ECOA does not define or mandate a Component Lifecycle Service; however it is clear that for any set of Components within a system to manage or be managed a common Service would need to be defined.

Since the Component Lifecycle Service is defined in the same way as any other ECOA Service, the System Designer is free to have as little or as much control as required. Indeed it would be possible to have different Component Lifecycle Services for different parts of the same system if various levels of control or management were required.

6.3 Example Management Hierarchy

Figure 1 shows an example management hierarchy containing multiple levels of managed components, grouped into a Managed Domain with multiple levels of management, along with some unmanaged components in an Unmanaged Domain. These Managed and Unmanaged Domains are purely conceptual and do not represent any actual artefacts.

The Manager Component in Management Level 1 is managing the two Components (Component A and Component B) in Management Level 2 through the use of a Component Lifecycle Service. Component A in Management Level 2 is providing functional behaviour along with managing one Component (Component C) in Management Level 3, again through the use of a Component Lifecycle Service.

The Component Lifecycle Services shown in Figure 1 may be the same for all of the managed Components, or could be varied based upon the particular requirements. For example the Component Lifecycle Service used between Management Levels 1 and 2 could be different to that used between Management Levels 2 and 3. Equally the entire system could make use of the same Component Lifecycle Service. For simplicity it is recommended to use the same Component Lifecycle Service across the entire system. In addition, using the same Component Lifecycle Service across the system, or even between systems, can allow a greater flexibility in re-use and re-deployment of Components.

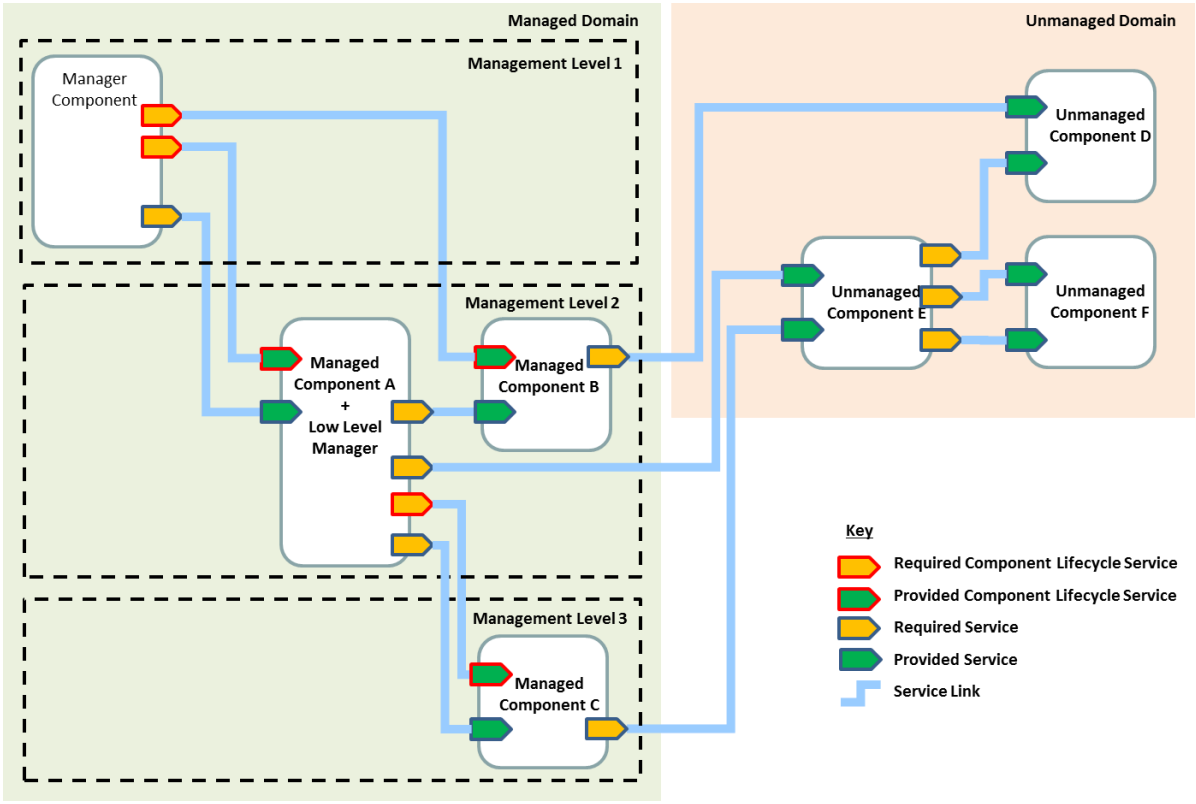

Figure 1 - Example Management Hierarchy

Because Components A, B and C are managed to some level their behaviour may be adjusted by their current 'state' or 'mode'

Components D, E and F are conceptually within the Unmanaged Domain and only provide functional Services to the rest of the system.

6.4 Examples of Component Lifecycle Services

6.4.1 Simple Example of a Component Lifecycle Service

A simple example of a Component Lifecycle Service could be defined as having two 'states' and three request/response operations:

- **States**
  - **IDLE**
  - **RUNNING**
- **Request/Response Operations**
  - **start**
    - **has a single output parameter returning the state when complete**
      - **IDLE**
      - **RUNNING**
  - **stop**
    - **has a single output parameter returning the state when complete**
      - **IDLE**
      - **RUNNING**
  - **get_current_state**
    - **has a single output parameter returning the current state**
      - **IDLE**
      - **RUNNING**

The state chart for this example is shown in Figure 2 and shows the two states along with the possible transitions.

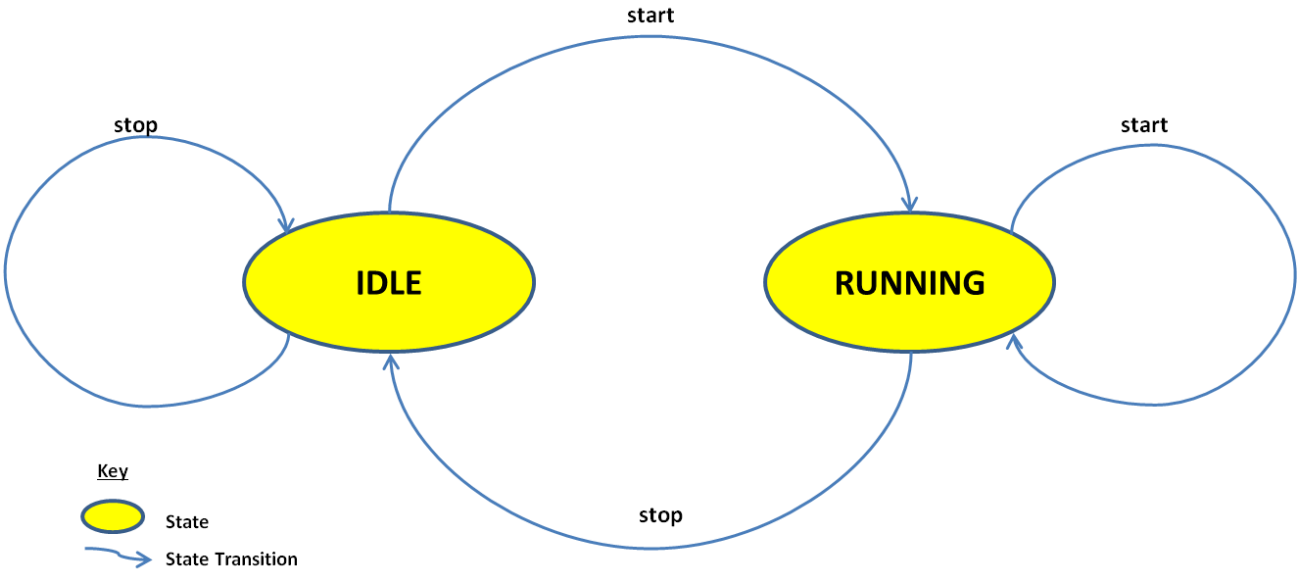

Figure 2 – State chart for Simple Component Lifecycle

As this is a very simple example there is no requirement for error detection in terms of requesting an invalid state change, as a start or stop will either transition to the other state, or remain in the current state.

This Component Lifecycle Service would be defined in the ECOA XML as follows:

```
<serviceDefinition
  xmlns="http://www.ecoa.technology/interface-2.0">

<use library="simple_lifecycle"/>

  <operations>
     <requestresponse name="start">
        <output name="state" type="simple_lifecycle:state"/>
     </requestresponse>

     <requestresponse name="stop">
        <output name="state" type="simple_lifecycle:state"/>
     </requestresponse>

     <requestresponse name="get_current_state">
        <output name="state" type="simple_lifecycle:state"/>
     </requestresponse>
  </operations>

</serviceDefinition>
```

This definition makes use of a predefined ‘simple_lifecycle:state’ type defined within a separate type library ‘simple_lifecycle’ as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.ecoa.technology/types-2.0">

     <types>
```

```
        <enum name="state" type="uint32">
            <value name="IDLE" valnum="0" />
            <value name="RUNNING" valnum="1" />
        </enum>
    </types>

</library>
```

Using the above Service Definition and associated type definition a manager Component would be able to send a ‘start’ or ‘stop’ request and receive a response with the state that the managed Component had transitioned to (or the state it had remained in). In addition the manager Component may, at any time, send a ‘get_current_state’ request and receive a response with the current state of the managed Component.

The implementation of this state chart within a Module of the managed Components would be trivial, as at initialization the state would be IDLE, and upon a request to ‘start’ it would transition to RUNNING once any required functionality was completed. In the same way upon a request to ‘stop’ whilst in the RUNNING state, it would transition to IDLE once any required functionality was completed. Any request to ‘start’ or ‘stop’ when in the ‘wrong’ state would just be ignored. Finally a request to ‘get_current_state’ would just respond with the current state of the Component.

This is a very trivial example of a Component Lifecycle Service, however it serves to illustrate that since Component Lifecycle Services are constructed using the normal ECOA Service rules, they are, as far as ECOA is concerned just another functional part of the system.

6.4.2 More Complex Example of a Component Lifecycle Service

The previous example of a Component Lifecycle Service is very simple; however it is possible to enhance the Component Lifecycle state charts to give the manager Components a better level of visibility of the managed Component state. In addition it is possible to build in some appropriate checks for invalid state transition requests.

In this case the Component Lifecycle Service will have four stable states and four transient states with a service that contains a single request/response operation and a single versioned data operation as follows:

- **States**
  - **IDLE**
  - **INITIALIZING**
  - **STOPPED**
  - **STARTING**
  - **RUNNING**
  - **STOPPING**
  - **FINISHING**
  - **FAILED**
- **Request/Response Operation**
  - **command_state_change**
    - **has a single input parameter with requested change**
      - **INITIALIZE**
      - **START**
      - **STOP**
      - **SHUTDOWN**
    - **Has a single output parameter with return status**
      - **OK**
      - **INVALID_STATE_REQUEST**
- **Versioned Data Operation**
  - **current_state – the current state of the Component**

The state chart for this example is shown in Figure 3 and shows the eight states along with the possible transitions.

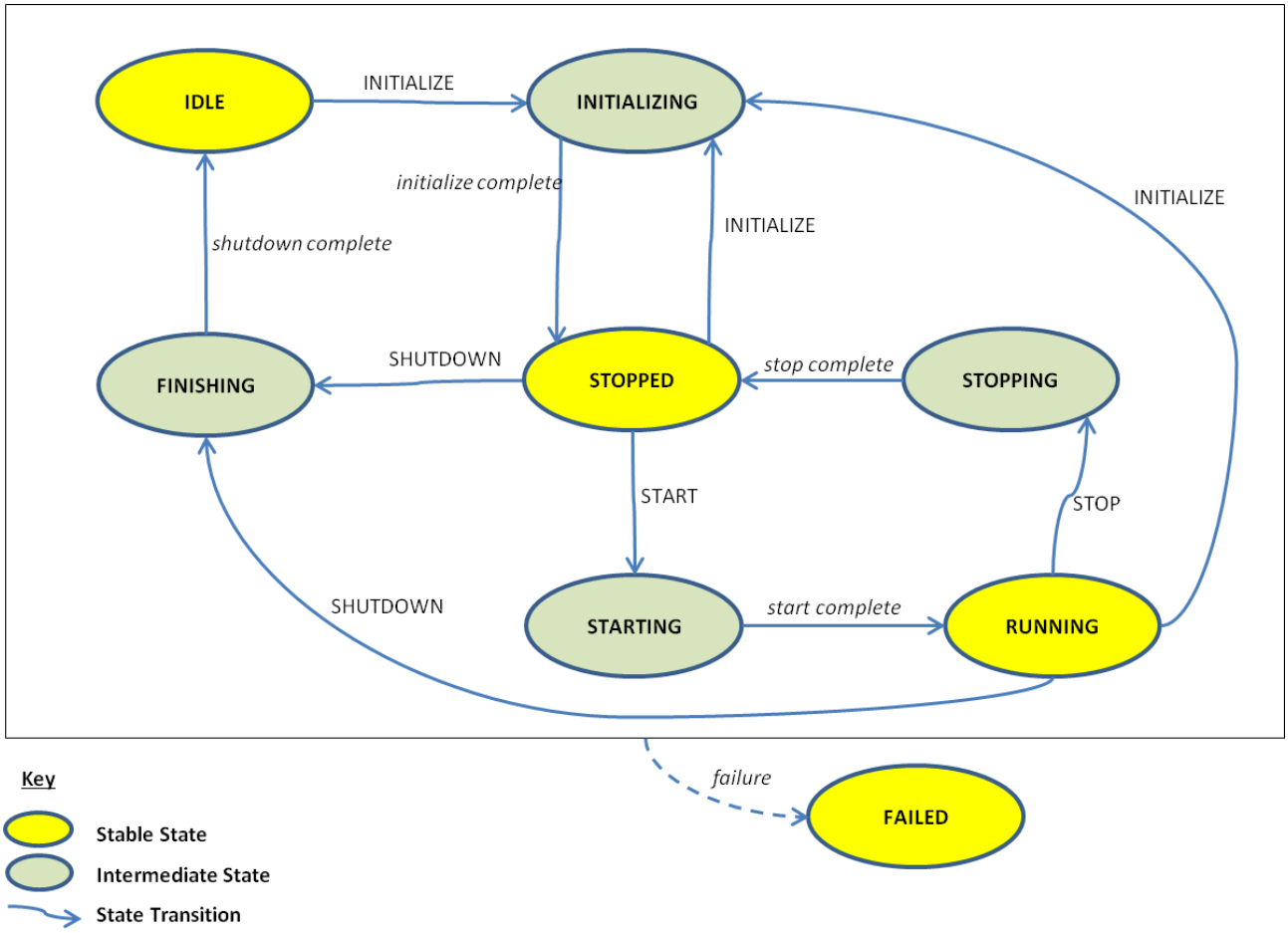

Figure 3 - State chart for more complex Component Lifecycle

The Component Lifecycle Service would be defined in the ECOA XML as follows:

```
<serviceDefinition
  xmlns="http://www.ecoa.technology/interface-2.0">

<use library="complex_lifecycle"/>

  <operations>
     <requestresponse name="command_state_change">
        <input name="commanded_state" type="complex_lifecycle:state_command"/>
        <output name="state" type="complex_lifecycle:status"/>
     </requestresponse>

     <data name="current_state" type="complex_lifecycle:state"/>
  </operations>

</serviceDefinition>
```

This definition makes use of a predefined types ‘complex_lifecycle:state_command’, ‘complex_lifecycle:status’ and ‘complex_lifecycle:state’ defined within a separate type library ‘complex_lifecycle’ as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.ecoa.technology/types-2.0">

     <types>
          <enum name="state_command" type="uint32">
               <value name="INITIALIZE" valnum="0" />
               <value name="START" valnum="1" />
               <value name="STOP" valnum="2" />
               <value name="SHUTDOWN" valnum="3" />
          </enum>

          <enum name="status" type="uint32">
               <value name="OK" valnum="0" />
               <value name="INVALID_STATE_REQUEST" valnum="1" />
          </enum>

          <enum name="state" type="uint32">
               <value name="IDLE" valnum="0" />
               <value name="INITIALIZING" valnum="1" />
               <value name="STOPPED" valnum="2" />
               <value name="STARTING" valnum="3" />
               <value name="RUNNING" valnum="4" />
               <value name="STOPPING" valnum="5" />
               <value name="FINISHING" valnum="6" />
               <value name="FAILED" valnum="7" />
          </enum>
     </types>

</library>
```

Using the above Service Definition and associated type definition a manager Component would be able to send a ‘command_state_change’ request and receive a response informing it that the request had either been accepted (‘OK’) or that it was an ‘INVALID_STATE_REQUEST.

If a valid ‘command_state_change’ request is received by a managed Component, then it would either respond with an ‘INVALID_STATE_REQUEST’, or change the Component state to the appropriate intermediate state (INITIALIZING, STARTING, STOPPING, FINISHING), and publish this as versioned data before beginning to functionally change the Component. Once the functional change was complete the managed Component would change its state to the appropriate stable state (IDLE, STOPPED, RUNNING, FAILED) and publish this as versioned data.

The implementation of this state chart within a Module of the managed Components would be more complex than the previous example as it has more states and behaviour associated with the management of functionality. In addition to generating valid state transitions upon request, the Module will also generate a response when an invalid request is made in a particular transition. For the state chart shown in Figure 3 the behaviour of a Component in response to a requested state change is shown in Table 1. NOTE that the ‘New States’ in the table are not all stable states, and therefore the Component would ultimately transition to a stable state based upon a valid request.

Table 1 - New States and Responses to Commanded State Requests

		Commanded State							
		INITIALIZE		**START**		**STOP**		**SHUTDOWN**	
Current State	**IDLE**	New State	INITIALIZING	New State	IDLE	New State	IDLE	New State	IDLE
		Response	OK	Response	ISR*	Response	ISR*	Response	ISR*
	INITIALIZING	New State	INITIALIZING	New State	INITIALIZING	New State	INITIALIZING	New State	INITIALIZING
		Response	ISR*	Response	ISR*	Response	ISR*	Response	ISR*
	STOPPED	New State	INITIALIZING	New State	STARTING	New State	STOPPED	New State	FINISHING
		Response	OK	Response	OK	Response	ISR*	Response	OK
	STARTING	New State	STARTING	New State	STARTING	New State	STARTING	New State	STARTING
		Response	ISR*	Response	ISR*	Response	ISR*	Response	ISR*
	RUNNING	New State	INITIALIZING	New State	RUNNING	New State	STOPPPING	New State	FINISHING
		Response	OK	Response	ISR*	Response	OK	Response	OK
	STOPPING	New State	STOPPPING	New State	STOPPPING	New State	STOPPPING	New State	STOPPPING
		Response	ISR*	Response	ISR*	Response	ISR*	Response	ISR*
	FINISHING	New State	FINISHING	New State	FINISHING	New State	FINISHING	New State	FINISHING
		Response	ISR*	Response	ISR*	Response	ISR*	Response	ISR*
	FAILED	New State	FAILED	New State	FAILED	New State	FAILED	New State	FAILED
		Response	ISR*	Response	ISR*	Response	ISR*	Response	ISR*

ISR* - INVALID_STATE_REQUEST

7 Fault Handling

7.1 Fault Handling Concept

7.1.1 Fault Handling Levels

ECOA defines the concept of Fault Handling at various levels; the Application, ECOA Infrastructure and underlying OS ([AS] Part 1).

Application level fault handling typically deals with the 'functional' view of Fault Handling, and is specific to the functional behaviour of the particular system. This type of fault handling makes use of domain knowledge in order to determine the best course of action when particular errors are detected.

ECOA Infrastructure level fault handling understands the ECOA architecture and the various mechanisms it uses. It will generally not understand the context in which it operates, and will allow a tailorable response to a particular detected error.

Underlying OS level fault handling is usually specific to the OS in question and in general it will not understand the context in which it operates. It would normally provide a generic capability that may be 'tailored' by the user.

The level of support for Fault Handling provided by the ECOA platform will be determined by the platform procurement requirements and therefore some fault handling mechanisms may not be available.

7.1.2 Fault Handling at the Application Level

This level of Fault Handling is outside the scope of the ECOA Standard, and as such is left to the System Designer/System Integrator to define.

ECOA does provide some low-level mechanisms to allow the Application Developer to communicate the detection of an error to the appropriate entity.

It allows:

- The reporting of detected errors from a Module to the ECOA Fault Handler
    - This may be fatal or non-fatal

This mechanism may be used by the Application Developer to report errors based upon its functional knowledge. Note that if a Module raises a fatal error, the ECOA infrastructure will shut down all modules in the Component.

7.1.3 Fault Handling at the ECOA Infrastructure Level

This level of Fault Handling is defined within ECOA as a set of generic errors based upon the ECOA mechanisms, and a set of possible recovery actions.

The ECOA Fault Handler may be implemented either as a specialised ECOA component, or as additional code integrated with the Container, which allows the functionality to be implemented in the most appropriate way. In either case the APIs are the same, and the functionality is defined by the System Designer/System Integrator.

Ultimately if the Fault Handler cannot handle any particular error it will have the option to report it to the underlying OS for further processing.

7.1.4 Fault Handling at the Underlying OS Level

This level of Fault Handling is outside the scope of ECOA and will be dependent upon the underlying OS and its capabilities. At this level fairly generic responses will be possible, and may not be directly associated with particular ECOA mechanisms or application functional behaviour.

7.2 The Fault Handler as additional Container code

The main advantage of implementing the Fault Handler as additional Container code is that it is very simple and does not require the ECOA Infrastructure to support a full Component implementation.

A very simple implementation would be to use a look-up table, where upon being invoked through the error notification API the Fault Handler would use the information to determine the recovery action required (if any) from a static table. If a recovery action is required, then the Fault Handler invokes the ECOA Infrastructure API to request the appropriate recovery action.

This simple implementation can be made more generic by allowing the look-up table to be defined through the use of a file, and would allow the implementation to be re-used in different systems by simply modifying the provided configuration file.

A disadvantage of this approach is that the Fault Handler is unable to take advantage of domain knowledge to influence its decision making. It has no functional context by which to vary its behaviour, and so by its very nature would be limited in its response to certain fault scenarios.

7.3 The Fault Handler as a Component

The main advantage of implementing the Fault Handler as a Component is that is uses the same concepts as a normal Component (e.g. Services/Operations etc.). This means that a Fault Handler implemented in this way is able to 'bridge the gap' between the Fault Handling at the ECOA Infrastructure level and that at the Application level.

A simple implementation can be made more generic by once again using a look-up table to define the required behaviour and using a file or the PINFO mechanism to change the table being used. In addition it would be possible to add some domain knowledge and current application state to the decision making and potentially make the system respond more appropriately to each fault scenario.

The possibilities when using a Fault Handler implemented as a Component are endless, as it could be designed as a fundamental part of the Application, not only invoking recovery actions through the ECOA Infrastructure API as required, but ultimately controlling or informing the Applications of faults within the system. Indeed a Fault Handler implemented as a Component could in effect be seen as a 'Manager' Component making use of Component Management (see section 6) to coordinate a change in system behaviour based upon errors reported from various entities within the system.

A disadvantage of this approach is that the Fault Handler can, if not carefully implemented, become very specific to an Application and ECOA Software Platform combination, thus making the reusability of the Fault Handler very limited. With careful thought however and the use of configuration tables it may be possible to implement a more generic and re-useable solution.

7.4 Interaction between the ECOA domain and the underlying OS

At any point the Container code is free to interact with the underlying OS layer in order to best manage the system. This may be default behaviour within the Container for scenarios where it cannot interact with the ECOA Fault Handler to report a detected error. In these cases it is the responsibility of the Platform Provider and System Integrator to define the behaviour such that it delivers the requirements of the System.

Additionally it is possible for the Fault Handler itself to interact as required with the underlying OS layer. This allows the provider of the Fault Handler to decide, based upon the reported errors, what the most appropriate course of action is.

In the case where the Fault Handler is implemented as additional Container code (section 7.2), the System Integrator may use the underlying OS layer APIs as required to fulfil the requirements.

In the case where the Fault Handler is implemented as a Component (section 7.3), then it may be prudent to isolate these interactions into one particular module within the Fault Handler, such that the remainder of it is still portable. In this case the Fault Handler would be operating as a Driver Component, and must comply with the requirements imposed by this.

For all cases it is the System Integrators responsibility to ensure that whatever behaviour is implemented it will satisfy not only the functional requirements of the system, but comply with any scheduling requirements to ensure the system behaviour is maintained appropriately.