European Component Oriented Architecture (ECOA®) Collaboration Programme: Guidance Document: Design Patterns

Date: 30/08/2017

Prepared by
Dassault Aviation

Contents

Figures

0 Executive Summary

This document provides examples of design patterns which may be considered by AS issue 6 users for developing ASCs. Each design pattern focuses on a different topic of interest.

1 Scope

This document is intended to provide examples of design patterns aligned with AS issue 6.

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 provides examples of design patterns.

2 Introduction

This document provides examples of design patterns which may be considered by AS issue 6 users for developing ASCs. Each design pattern focuses on a different topic of interest.

3 Abbreviations

API	Application Programming Interface
ASC	Application Software Component
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
FR	French
IAWG	Industrial Avionics Working Group
I/O	Inputs-Outputs
OS	Operating System
PF	Platform
QoS	Quality of Service
RR	Request-Response
STD	Standard
TR	Technical Report
TRL	Technology Readiness Level
UDP	User Datagram Protocol
UK	United Kingdom
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 Design Patterns

6.1 Design patterns for ASC implementation

The following design patterns are examples of possible ways of implementing an ASC using one or more modules. These patterns are not exhaustive and are provided for illustration only.

6.1.1 Single Module Component Design Pattern

Figure 1 illustrates what is believed to be the “default” design pattern, which is sufficient when there is no specific software development requirement to split the functionality into multiple modules (such as parallel computation) within the ASC. Having a single Module greatly simplifies application code as it eliminates the need for functional synchronization between several Modules. This Module Instance (M1) is automatically started by the ECOA Platform.

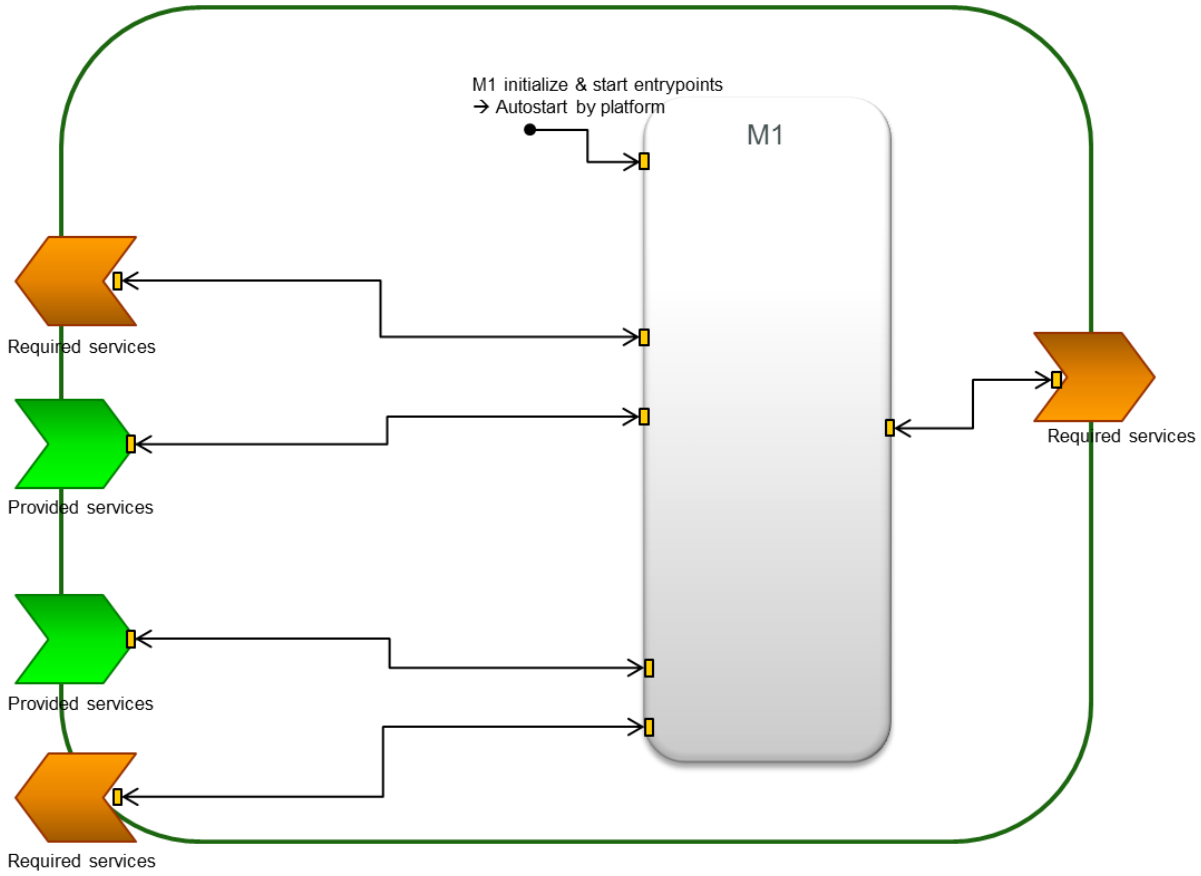

Figure 1: Example design pattern – Single Module ASC

6.1.2 Multiple Module Component Design Pattern

Figure 2 illustrates a pattern for implementing multiple Module ASCs when there is a requirement for performing parallel computation. It features a centralized architecture with a “functional manager”. From an ECOA point of view, this “functional manager” is a Module Instance without any privilege (no difference with other Module Instances). All Module Instances are automatically started by the ECOA Platform (in terms of ECOA technical Module lifecycle management). The “functional manager” controls the functional initialization of the ASC once all Module Instances are in “RUNNING” state from an ECOA lifecycle point of view. This involves normal functional interactions between Module Instances. For instance, Module Instances may tell the “functional manager” about their functional initialization state (e.g: not initialized / initialization in progress / initialized…) and the “functional manager” would request Module Instances to perform functional initialization. It may synchronize ASC level states based on the functional initialization state from Module Instances.

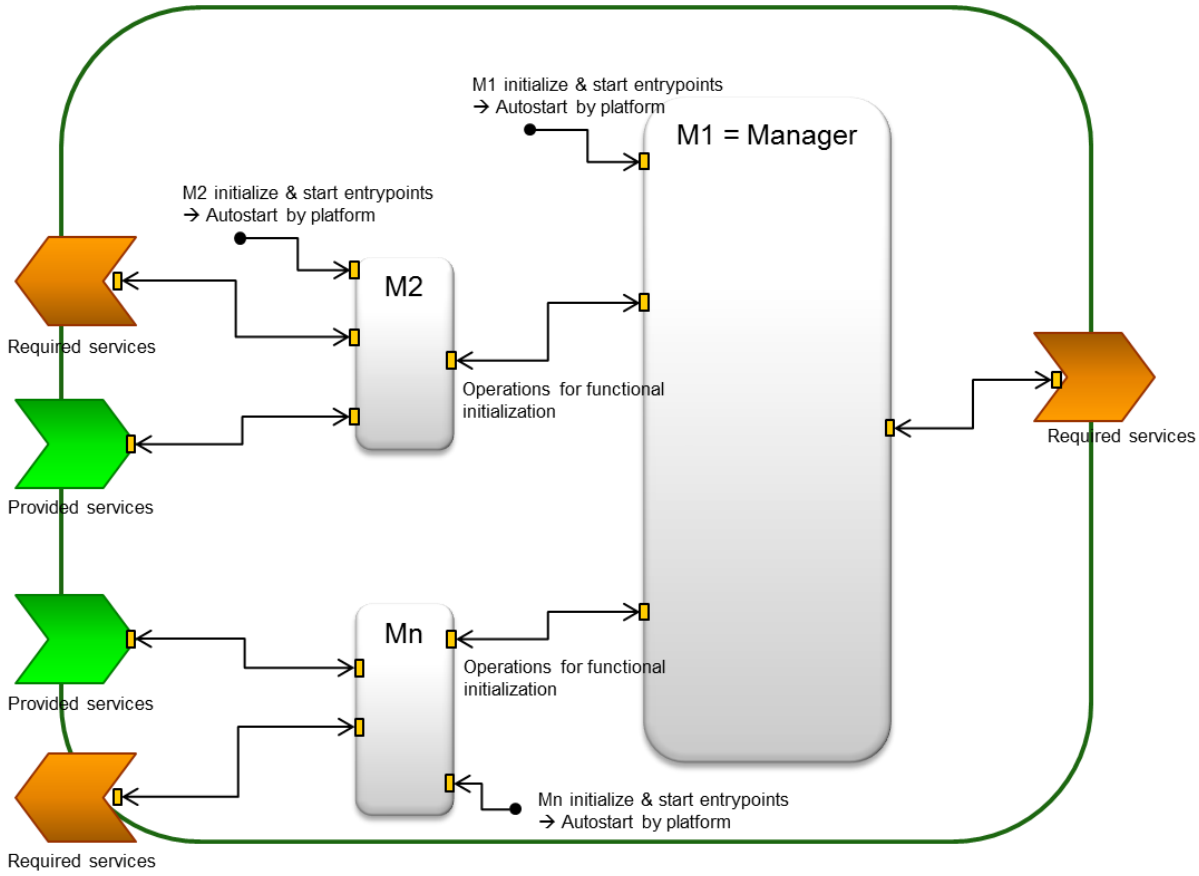

Figure 2: Example design pattern – « Centralized Decision & Parallel calculation » template: multi-modules ASC + centralized management

6.1.3 Multiple Module Component with Parallel Execution Design Pattern

Figure 3 illustrates a pattern for implementing multiple Module ASCs when there is a requirement for performing parallel computation. The difference with the previous design pattern is that it does not feature any “functional manager”. All Modules initialize functionally as soon as they are running, and they do not need to synchronize functionally with each other. The use of this or the previous pattern depends on the functional requirements of the Component.

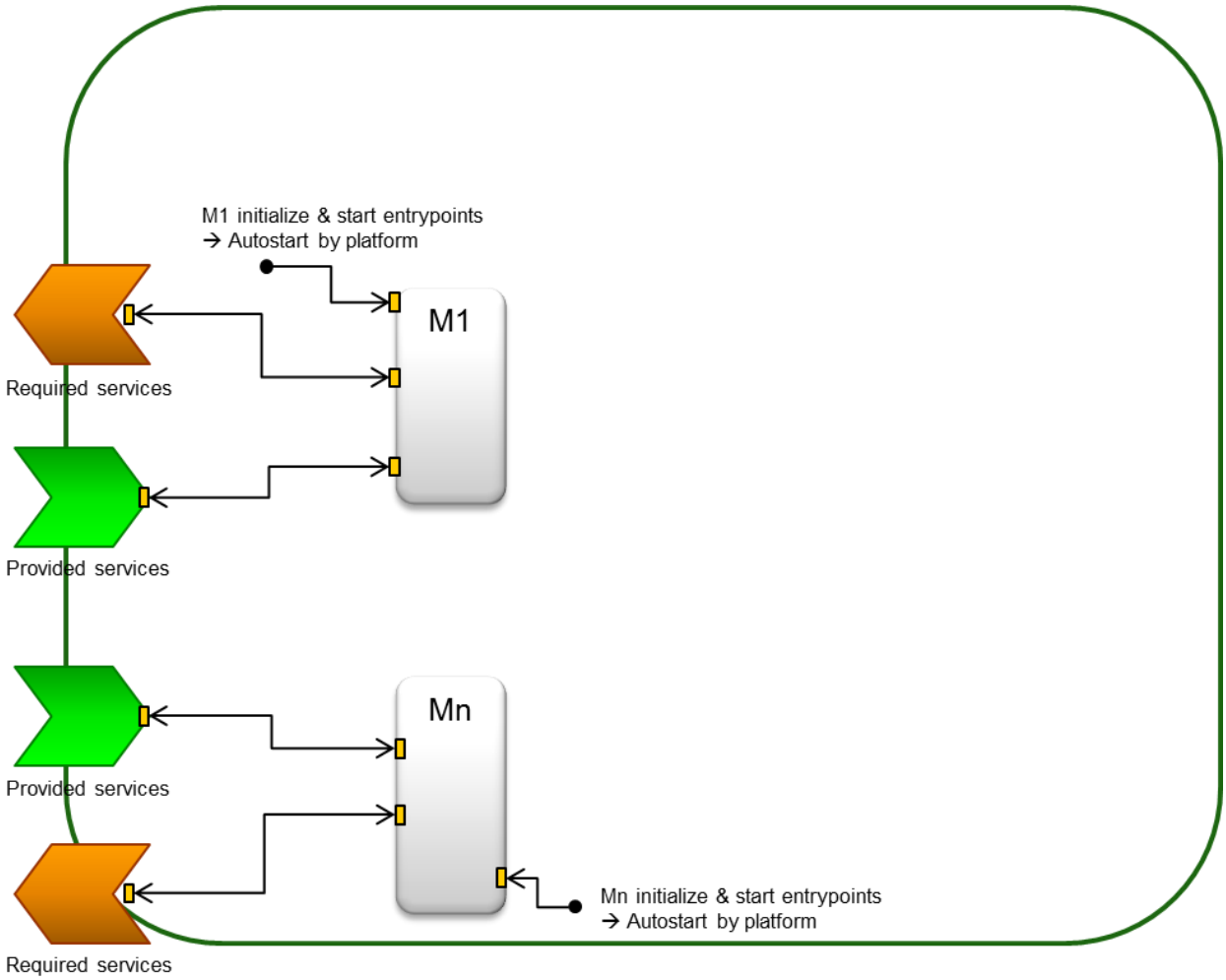

Figure 3: Example design pattern – « Decentralized Decision & Parallel calculation » template: multi-modules ASC + decentralized management

6.2 Design pattern for service availability

6.2.1 "Passive" or "pro-active" service availability management

"Pro-active" service availability management implies that all components have to monitor themselves and provide the availability of their services, so that this information is received by all their clients.

Another approach is to rely on "passive" service availability. With this approach a client monitors the availability of the services it requires, based on observing the functional behaviour of these services. This has several advantages over the "pro-active" pattern:

- There is less network traffic
- It is simpler as it works with existing APIs, not by creating a dedicated one.
- No component is monitoring itself, making the system more robust.
- Availability of a service can be different for different clients, the network path is accounted for.
- "Passive" and "pro-active" can be mixed and matched. "Pro-active" service availability would be applied only in the places in the system where it is necessary.
- "Passive" service availability is focused on defensive programming and how to achieve it - which is necessary anyway even with "pro-active" service availability.

6.2.2 Design pattern for “pro-active” service availability management

In use cases where a “pro-active” service availability management is required, “normal” ECOA operation links provide everything needed by the ASC Supplier to manage “pro-active” service availability:

- In terms of service definition, it means that service availability could be declared as an explicit versioned data operation in services where it is actually needed. This allows for traceability between software implementation and system specification where such service availability may be explicitly declared as an interface.
- In terms of updating service availability, besides the functional application code which calculates the availability of a service, the ASC supplier writes that versioned data operation. This is done by getting a handle on the data, writing it and publishing it.
- In terms of reading the availability of required services, the application code reads a versioned data operation. This is done by getting a handle on the data, reading it, then releasing the handle on the data.
- When service availability is implemented as a periodic versioned data, monitoring whether this data has become stale allows client ASCs to detect that services have become unavailable or that there is a network issue. This implies the implementation of such data monitoring code in the application. Such defensive programming on the application side may improve its robustness to network failures (either temporary or permanent).
- In addition, using service operations allows any module to write and read service availability. This enables a decentralized ASC architecture to be used in places where a centralized design pattern is not appropriate, eliminating application code related to functional synchronization between a manager and other modules. This provides the benefit of flexibility in options for implementing ASCs.

The following design patterns illustrate the above principles. These patterns are not exhaustive and are provided for illustration only.

Figure 4, Figure 5 and Figure 6 illustrate a pattern for managing service availability, which could be used within an ECOA Composite under the responsibility of a unique application supplier. The supplier could optimize the internal design of the Composite by managing services availability “globally” through a single availability state data per ASC. This pattern is illustrated for each of the three patterns about ASC implementation. These figures show an additional operation which illustrates the change a component would require if an externally provided service was updated to include an availability service operation.

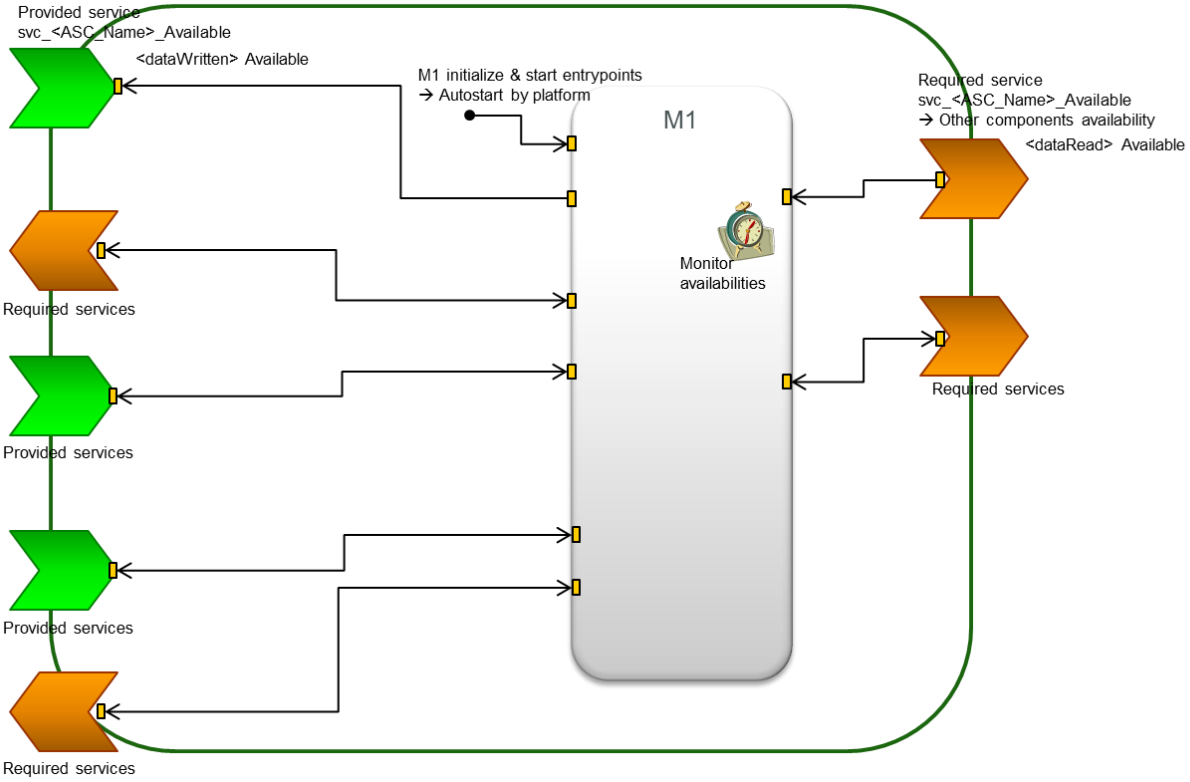

Figure 4: Example design pattern – Global svc availability @ASC level, illustrated on template 1) « Single Module ASC »

Figure 5 illustrates the example of ASC implementation pattern #2 (centralized architecture), where one Module Instance has the role of a “functional manager” which controls the functional states and modes of the ASC, including functional initialization. This “functional manager” is also responsible for setting the ASC availability information and for monitoring availabilities from other ASCs via versioned data.

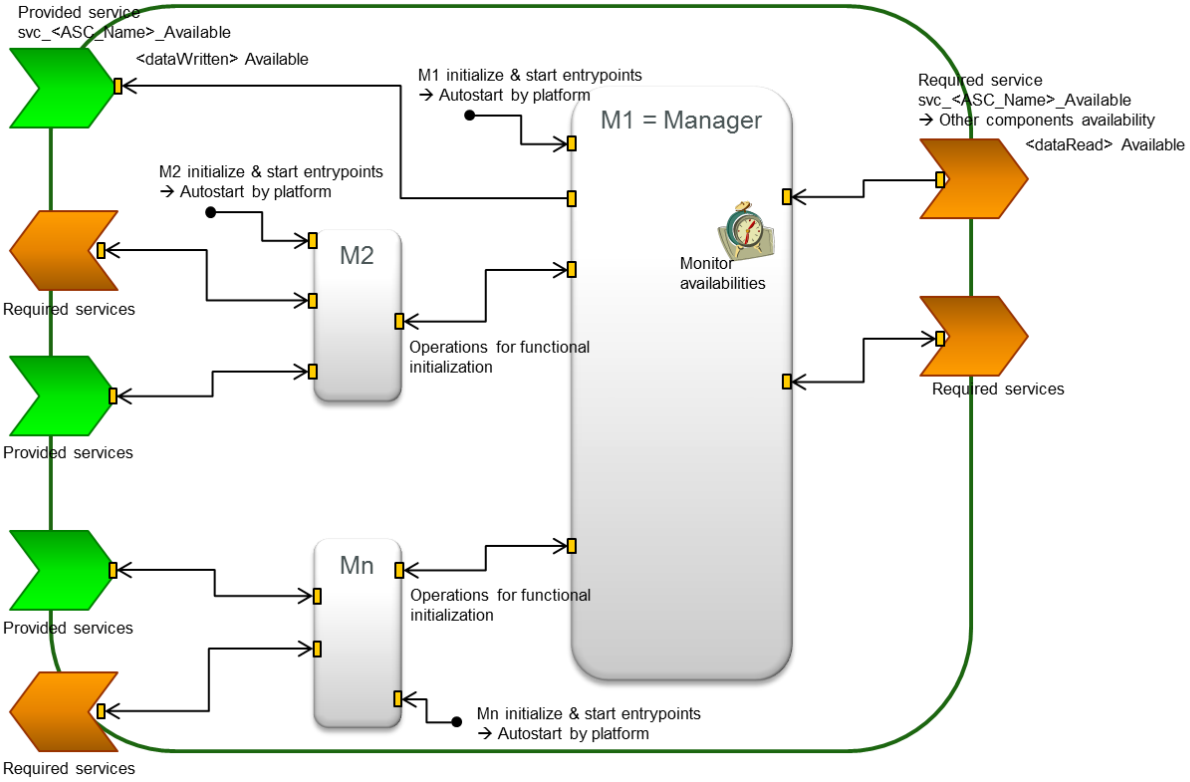

Figure 5: Example design pattern – Global svc availability @ASC level, illustrated on template 2) « Centralized Decision & Parallel calculation »

Figure 6 illustrates the design pattern on the example of ASC implementation pattern #3 (decentralized architecture).

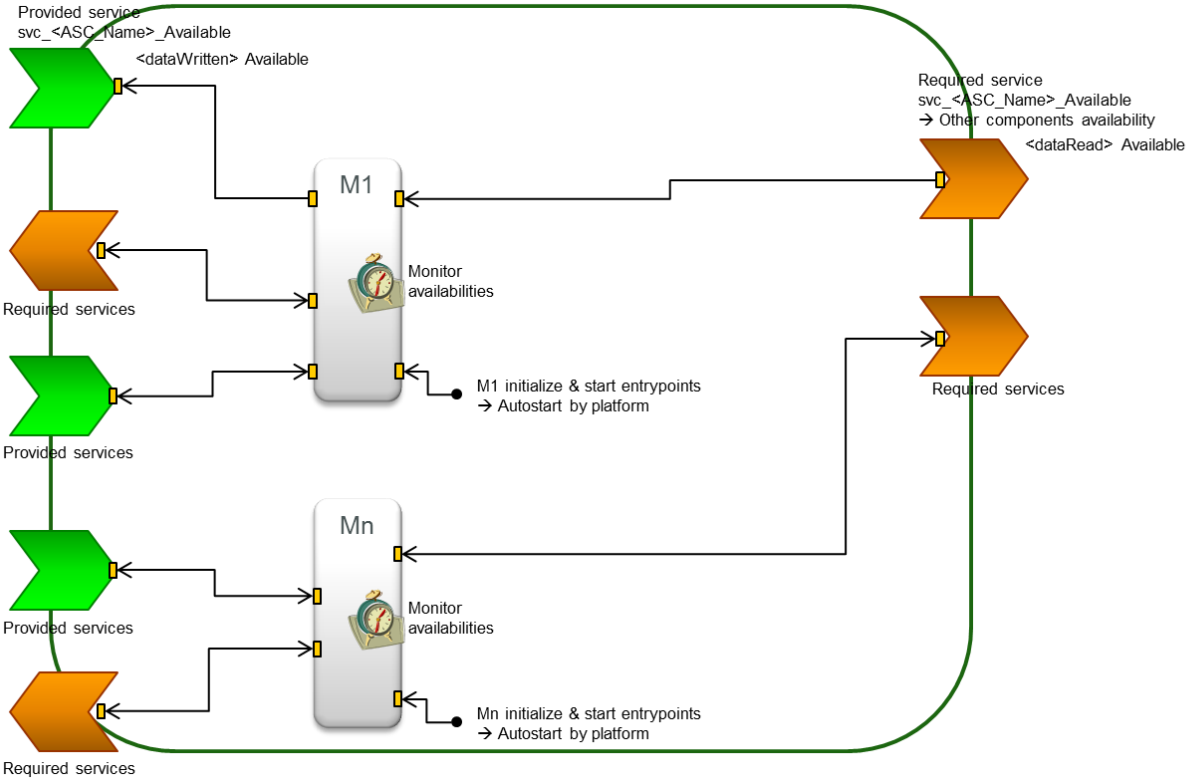

Figure 6: Example design pattern – Global svc availability @ASC level, illustrated on template 3) « Decentralized Decision & Parallel calculation »

Figure 7, Figure 8 and Figure 9 illustrate a pattern for managing service availability, which could be used in a service oriented design mixing ASCs from different suppliers. This pattern is illustrated for each of the three patterns about ASC implementation into Modules.

Only in services that functionally require availability information would the supplier read or write a versioned data conveying that information.

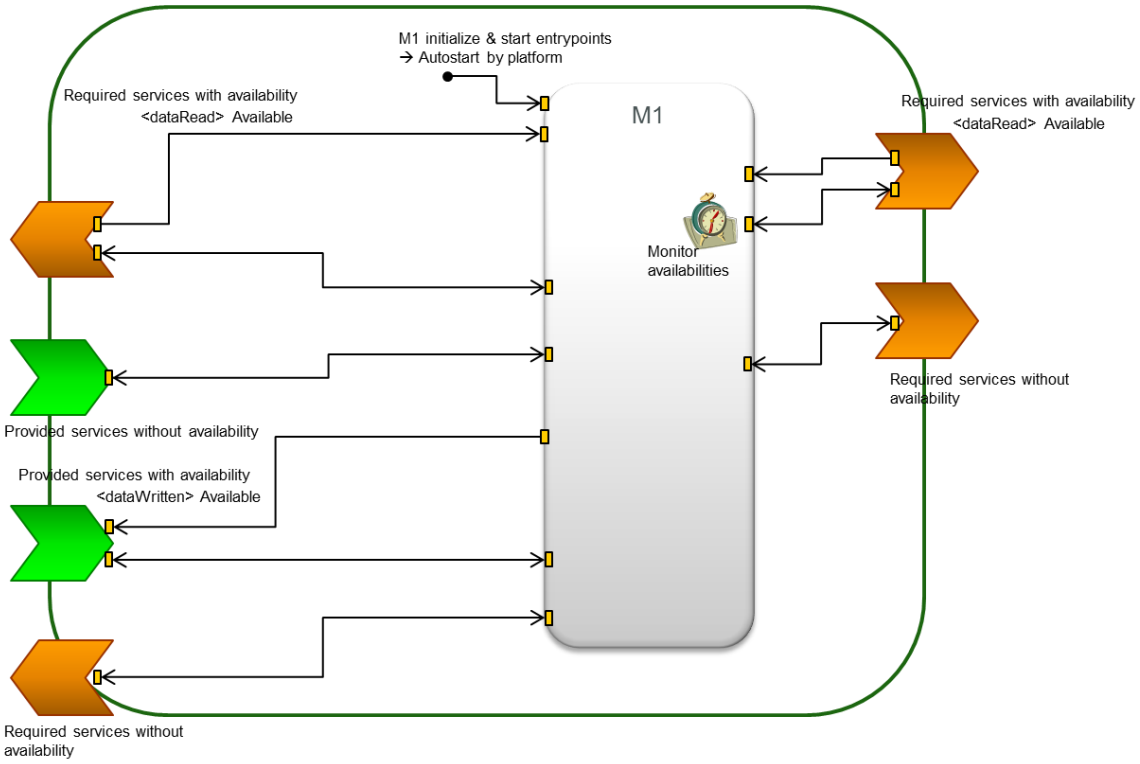

Figure 7: Example design pattern – Availability @service level for some services, illustrated on template 1) « Single Module ASC »

Figure 8 illustrates the example of component implementation pattern #2 (centralized architecture), where the “functional manager” is responsible for writing the availability versioned data of provided services which feature such information. It is also responsible for monitoring the availability versioned data of required services depending on whether it needs to do so functionally.

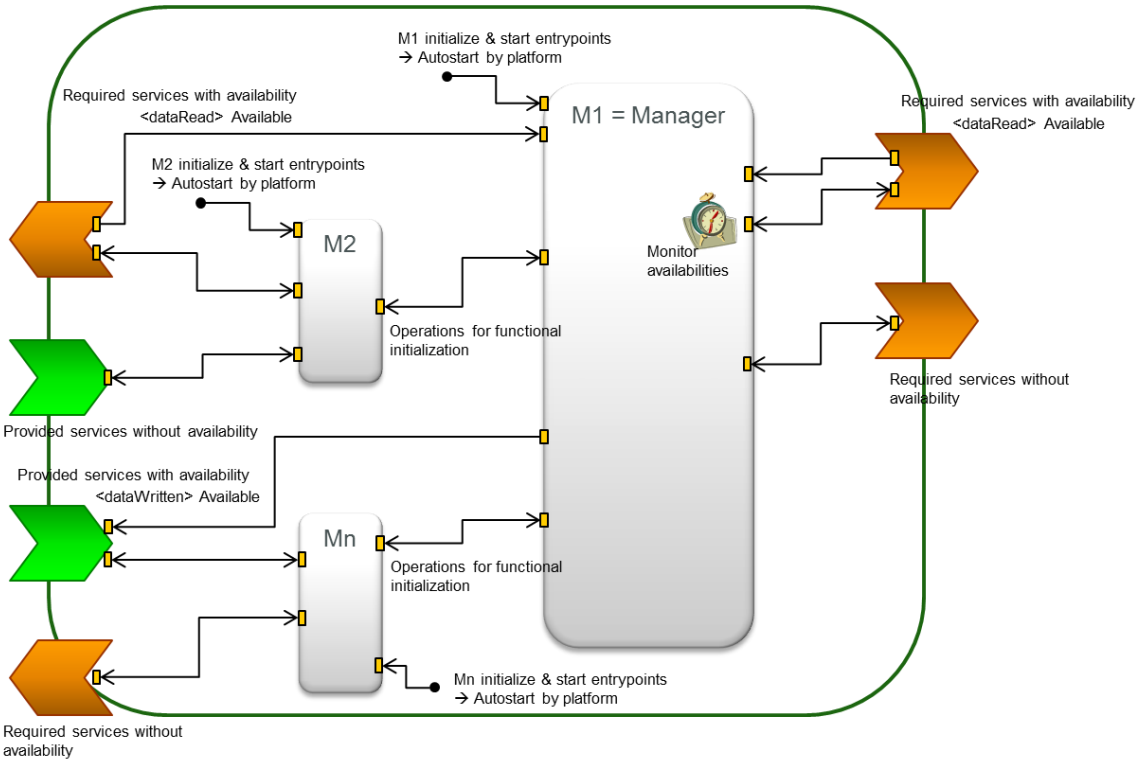

Figure 8: Example design pattern – Availability @service level for some services, illustrated on template 2) « Centralized Decision & Parallel calculation »

Figure 9 illustrates that service availability information could also be managed in a decentralized way provided no functional synchronization is required between Modules.

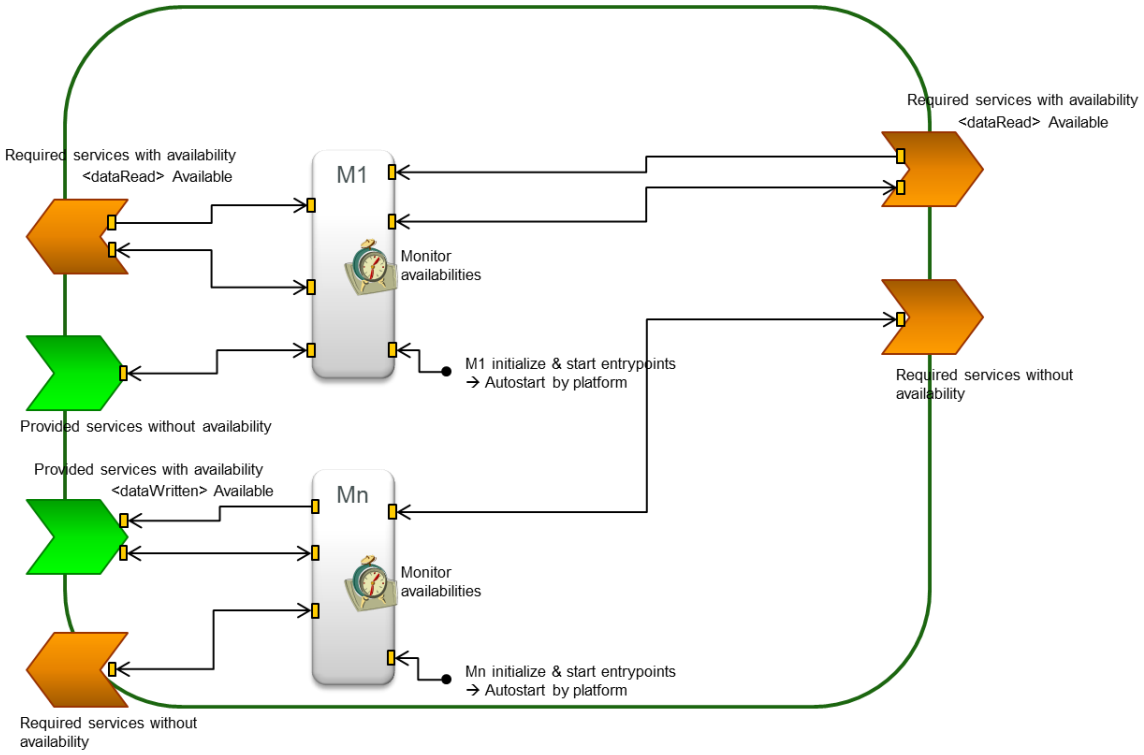

Figure 9: Example design pattern – Availability @service level for some services, illustrated on template 3) « Decentralized Decision & Parallel calculation »

6.3 Design patterns for handling redundant service providers

Figure 10 and Figure 11 illustrate example design patterns of possible ways for managing redundant service providers.

These patterns are not exhaustive and are provided for illustration only. Redundant service providers and the design patterns using them may also be employed to provide data correlation or data fusion between the providers

In thie pattern described in Figure 10, the “SensorManager” ASC features two required service instances, each of them being connected to one provided service instance.

An explicit versioned data operation (called “Available” in this example) may be added in the service to monitor service availability, as illustrated previously in section 6.2. This is not mandatory and depends on the functional nature of the service, as any existing periodic data (e.g. AV State vector) could be used for monitoring the service availability.

The “SensorManager” ASC monitors the periodic versioned data that conveys the service availability information in both required service instances.

The “SensorManager” ASC has the freedom and flexibility for deciding how to use these redundant services:

- It may use data from just one provider and ignore the other,
- Or it may use data from both providers in order to consolidate the AV State vector and detect any abnormal divergence between them.

The “SensorManager” ASC is responsible for resetting and managing any functional protocol with the provider(s) of its choice.

A third required service instance is featured for growth potential. It can be left unconnected and the “SensorManager” ASC may be configured with a property that instructs it to ignore it. The application code of the “SensorManager” ASC can be written in such a way that it will comply with the varying number of providers. The difference in terms of ECOA XML data is not significant.

The design pattern in Figure 10 is believed to be a flexible and versatile option as it covers all use cases.

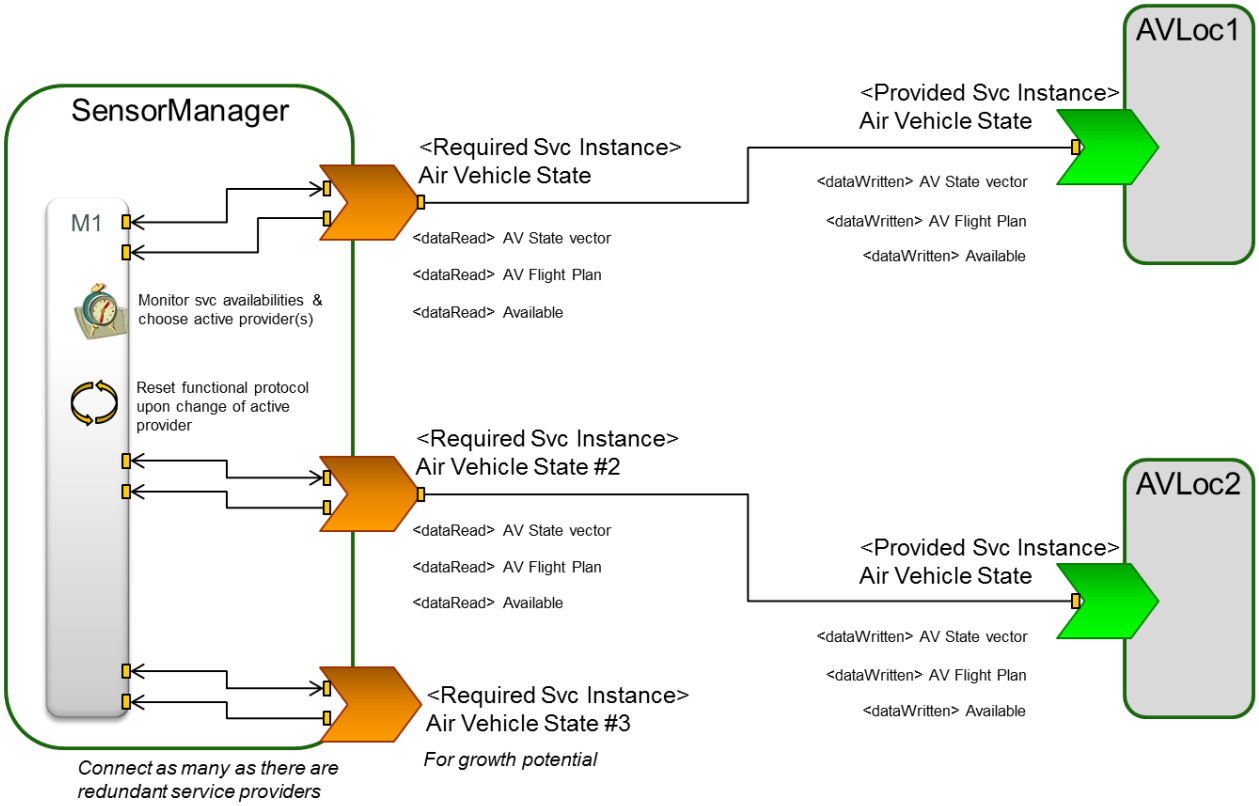

Figure 10: Example design pattern – Redundant service providers

In the design pattern illustrated in Figure 11, a “broker” ASC is inserted between the “SensorManager” ASC and the redundant “AVLoc” service providers.

The role of the broker is to perform the switch between redundant providers. The broker only exposes one provided service instance to the “SensorManager”.

A versioned data is added in the service exposed by the broker. When defined as “notifying” in “SensorManager” implementation XML, it enables the “broker” to be notified upon a change in the active provider and trigger any functional behaviour such as resetting functional protocol with the new active provider.

This pattern provides abstraction in terms of the number of service providers, as the SensorManager component in Figure 11 would not require updating if the number of service providers changed. A single “broker” may also be used to expose one service instance to any number of clients such as “SensorManager”.

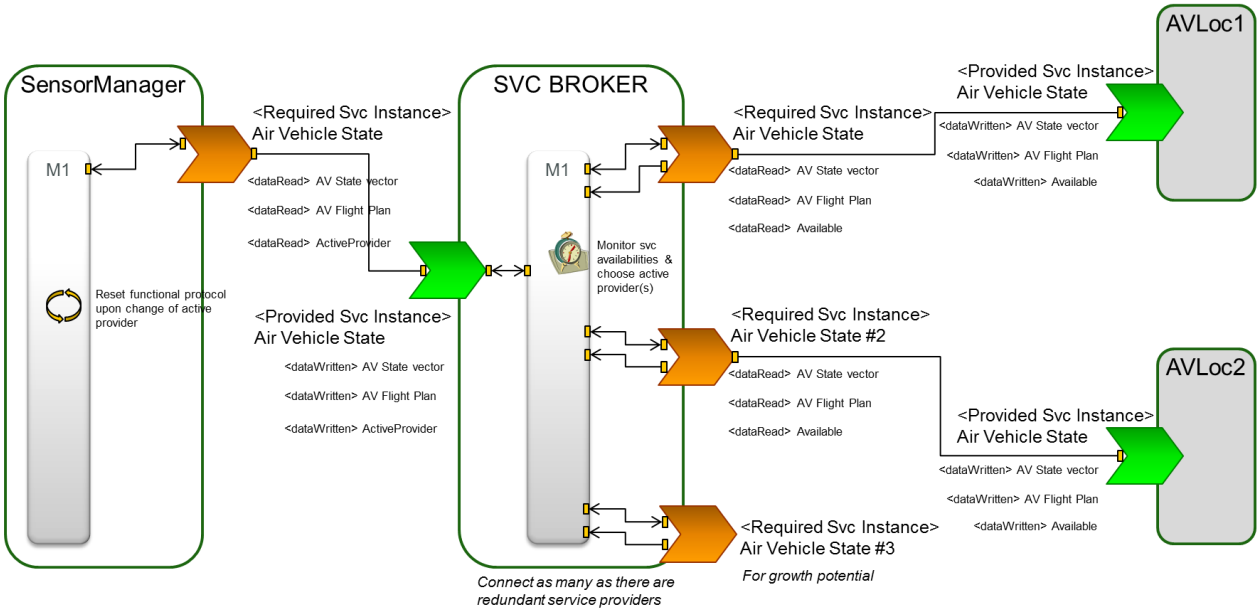

Figure 11: Example design pattern – Broker ASC

6.4 Design patterns for module FIFO management

Module FIFO management can be tailored to provide a fine level of control over the queueing of operations. The simple mechanism for FIFO management is to connect all sources to a single OperationLink, as illustrated by Figure 12. In this scenario, all three clients share the same fifoSize=”10” and activating=”true” settings.

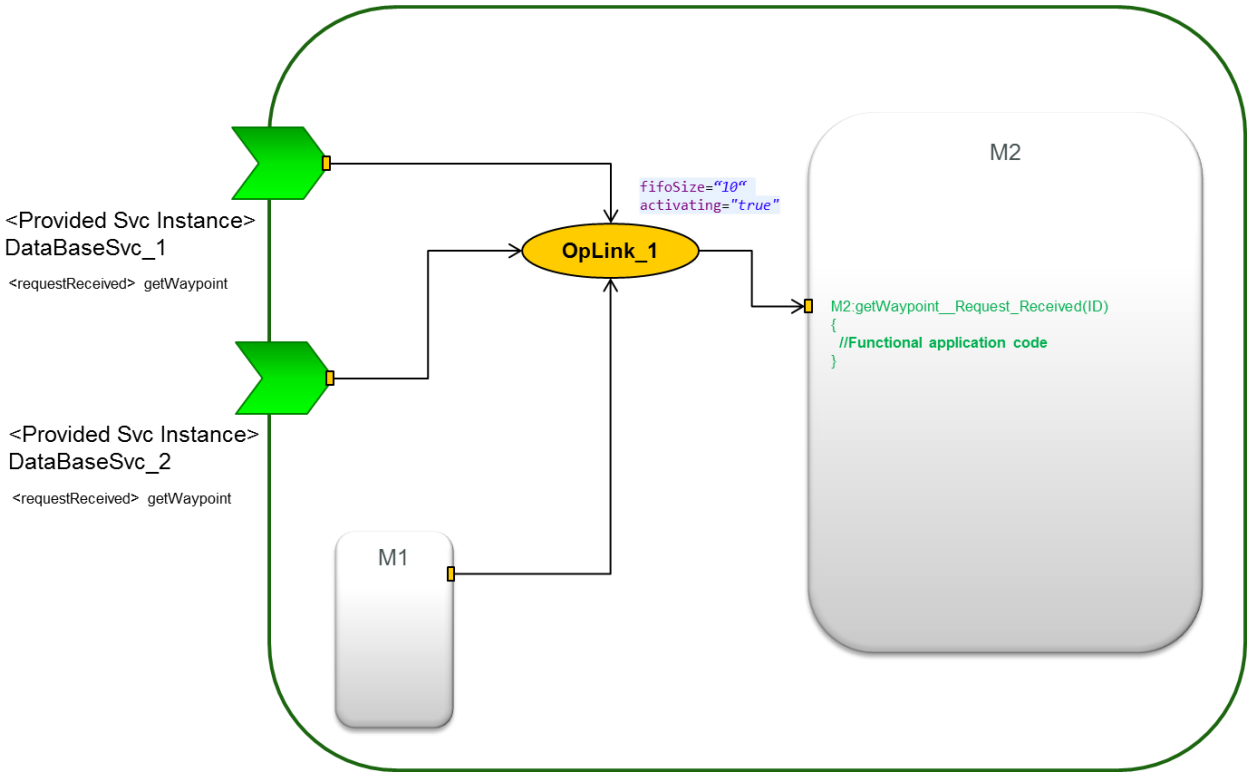

Figure 12: No module FIFO size management

6.4.1 Design pattern for Fine Level Control of Module FIFO management

The purpose is to manage module FIFO size and operation activation in case of multiple sources sending events or requests which are processed by the same module. The goal is to be able to set different module FIFO sizes and activation policies depending on the source of the event or request. This aims at mitigating the effects of a source which would send too many events or requests, thereby preventing events or requests from other sources being processed, which would happen if there was a single module FIFO size.

The design pattern consists of keeping the same operation links, but duplicating the Module entry points to match the number of operation links. In each of these duplicate entry points, the incoming call is forwarded to the same M2 internal function which contains the same application code as in Figure 12. This means that the Container only has to manage one FIFO per Module Operation entry point whilst it is still possible to assign different module FIFO sizes per source of incoming calls.

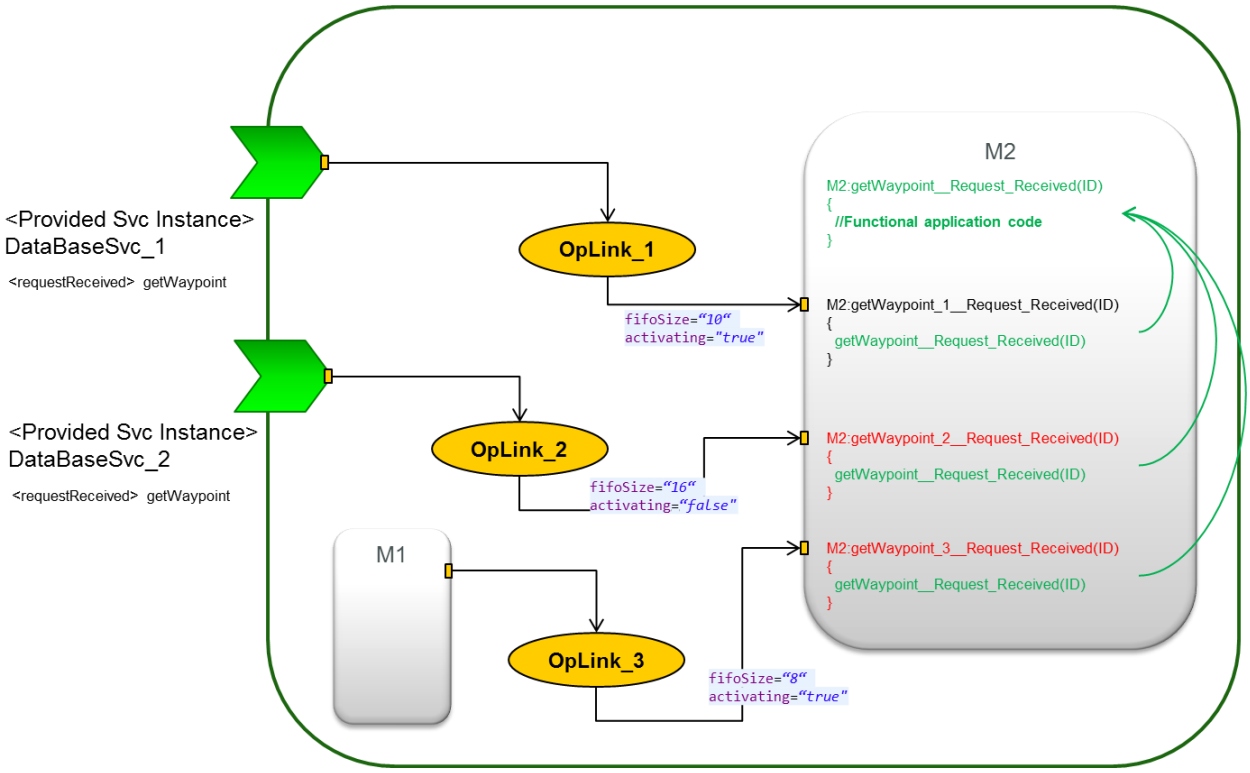

Figure 13: Module FIFO size management

Of course, this pattern should only be used if there is such a requirement; otherwise it is simpler to connect all sources to a single OperationLink as illustrated by Figure 12.