PYRAMID Exploiter's Pack Version 4.1
Annex B – Deployment Guide Issue 4.1

Ministry of Defence

This document has been prepared, as part of the PYRAMID Exploiter's Pack, in order to set out a generic approach to implementation of the PYRAMID Architecture. The PYRAMID Reference Architecture has not been created for any specific system. It is the user's responsibility to ensure that any article created using this document meets any required operational, functional and safety needs. The Author accepts no liabilities for any damages arising due to a failure of the user to verify the safety of any product produced using this document, nor for any damages caused by the user failing to meet any technical specification.

For further information regarding how you can exploit PYRAMID on your project, provide feedback following your review of the PYRAMID Exploiter's Pack V4.1, or have a technical query that you would like answering, please contact the PYRAMID Team using the following email address. PYRAMID@mod.gov.uk

This publication is available at https://www.gov.uk/government/publications/pyramid/pyramid-programme

EXECUTIVE SUMMARY

The MOD’s PYRAMID programme introduces a change to the current method of avionic systems design and procurement, aiming to make the next generation of air systems affordable, capable and adaptable by the adoption of an open architecture approach and systematic software reuse.

This document outlines how the system and software design processes and strategies used to develop air system software should be adapted when using the PYRAMID Reference Architecture (PRA), and how PRA artefacts could be implemented by an Exploiting Programme such that the Key User Requirements (KURs) can be realised. It identifies at what stage in the design and development lifecycle the different PRA artefacts are most useful. The development stages discussed in this document, along with associated approaches and the typical outputs, are as follows:

- Platform Independent System Design
    - **Build Sets**, defining which components are required to meet the customer scenario;
    - **Component Specifications**, defining the services provided and consumed by components;
    - **Bridges**, defining the interrelationships between components;
    - **Service Modelling** and **Data Modelling** approaches to defining Component Specifications and bridges.
- Platform Independent Component Design
    - A **data driving model**, representing how data driving is expected to be supported;
    - **Extensions**, defining how component extensions may be used;
    - **Classes**, defining the static structure of the component;
    - **States**, defining the dynamic behaviour of the component.
- Platform Specific System Development
    - An **infrastructure definition**, specifying the Execution Platform to support the components;
    - A **partitioning approach**, defining how components are separated or aggregated;
    - **Configuration**, to prepare data that supports the data driving design decisions;
    - A **component services approach**, defining the underlying mechanisms between services.
- Platform Specific Component Development
    - **Component Implementation**, which is software development of the component.

PRA policies are referenced throughout this document, and should be used to support tailoring of a PRA deployment.

CHANGE HISTORY

Date	Issue	Description of Changes
November 2019	1	First Issue.
January 2020	1.1	Embodiment of customer review comments.
December 2020	2	Document restructured to remove content duplicated across multiple documents. The approaches to defining component specifications has been expanded in the sections Service Modelling (Section 2.5) and Data Modelling (Section 2.6). New section has been added to capture best practice recommendations (Section 2.7). In addition changes have been made to this document resulting from query answering from both internal and external stakeholders, and general maturity improvements.
December 2021	3	Section 4.2.4 updated based on HMI changes. Material removed from this document to avoid duplication with the Component Connections PRA Policy. In addition changes have been made to this document resulting from answering queries from both internal and external stakeholders, and to improve maturity.
October 2022	3.1	Issue 3.1 is a UK OFFICIAL version of Issue 3. No changes to the content of the document have been made other than non-technical changes due to the up-issue of the PYRAMID Exploiter's Pack from Version 3 to Version 3.1 (headers/footers, references, etc.).
December 2022	4	The description of logical architecting diagram types in section 2.2 has been refined. Clarity improvements have been made to sections 2.3, Component Specification, and 2.5, Service Modelling. Good practice section 2.7.3 has been added to provide guidance on which component is most appropriate for managing certain calculations. The considerations around the rely-guarantee method in section 3.2.5 have been refined. In addition general maturity improvements have been made throughout this document.

Date	Issue	Description of Changes
September 2023	4.1	The document has now been updated due to now being released via Open Government License v3.

List of Effective Pages

82 pages UK OFFICIAL

82 pages in total

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

No tables contained in this document.

TERMS AND ABBREVIATIONS USED IN THIS DOCUMENT

Definitions of project terms, the meaning of acronyms and the meaning of abbreviations used in this document can be found in the PYRAMID Glossary Ref. [5].

REFERENCES

The reference numbers are consistent across all the documents in the PYRAMID Exploiter's Pack. This means that in this document, when a reference is not used, the corresponding reference number will not appear in the reference list.

Project Related Document References:

For the avoidance of doubt, all documents referenced below which form part of the PYRAMID Exploiter's Pack are subject to the terms of DEFCON 703.

Reference	Title, Document Number, Issue & Date
[2]	PYRAMID Exploiter's Pack Annex A: PRA Description Document, RCO_FUT_23_005, Issue 4.1, September 2023.
[4]	PYRAMID Exploiter's Pack Annex C: Compliance Guide, RCO_FUT_23_007, Issue 4.1, September 2023.
[5]	PYRAMID Exploiter's Pack Annex D: Glossary, RCO_FUT_23_008, Issue 12.1, September 2023.
[60]	Dstl, Security Guidance for PYRAMID Exploiters, DSTL/TR111125, Issue 1.0, October 2021.

Non-Project Related References:

Non-Project related references below contain information which is proprietary to that referenced third party. Any information from this source is subject to separate rights and terms.

Reference	Title, Document Number, Issue & Date
[8]	J. Borky and T. Bradley, Effective Model-Based System Engineering, 2019.
[9]	C. Raistrick et al, Model Driven Architecture with Executable UML, 2004.
[11]	T. Erl, Service-Oriented Architecture: Concepts, Technology and Design, 2005.
[12]	ISO/IEC 18384, Reference Architecture for Service Oriented Architecture (SOA RA), 2016.
[13]	C. Raistrick, Land Data Model Methodology and Modelling Standard, 2019. [Online]. Available: https://landopensystems.mod.uk
[21]	Airworthiness Security Process Specification, RTCA DO-326A, 2014.
[24]	M. Page-Jones, Practical Guide to Structured System Design, 1988.
[26]	The Open Group, Technical Standard for Future Airborne Capability Environment, 3.1 ed., 2020.
[27]	The Open Group, "FACE Shared Data Model," 2020, [Online]. Available: https://www.opengroup.org/face/docsandtools.
[28]	Multilateral Interoperability Programme, "MIP Information Model," [Online]. Available: https://www.mimworld.org.

Reference	Title, Document Number, Issue & Date
[29]	EUROCONTROL, “Aeronautical Information Exchange Model”.
[30]	Society of Automotive Engineers, “Unmanned Systems (UxS) Control Segment (UCS) Architecture: Data Model,” 2016.
[31]	RTCA DO-332, Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, Radio Technical Commission for Aeronautics, 2011.
[32]	RTCA, “Handbook for Object-Oriented Technology in Aviation (OOTiA),” Radio Technical Commission for Aeronautics, 2004.
[33]	Assurance Case Working Group, “Goal Structuring Notation Community Standard (version 2),” GSN Working Group. [Online]. [Accessed 24 10 2019].
[34]	R. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship,” 2008.
[35]	F. Devos, “Patterns and Anti-Patterns in Object-Oriented Analysis,” 2004. [Online]. Available: http://www.cs.kuleuven.ac.be/publicaties/doctoraten/cw/CW2004_06.pdf. [Accessed 2020].
[36]	S. S. a. T. A. M. Eileen A. Bjorkman, “Using Model-Based Systems Engineering as a Framework for Improving Test and Evaluation Activities,” *Systems Engineering,* vol. 16, no. 3, 2012.
[37]	SAE International, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, ARP-4761, 1996.
[38]	SAE International / EUROCAE, Guidelines for Development of Civil Aircraft and Systems, ARP-4754A, 2010.
[39]	F. D. Giraldo, Introduction to Model-Driven Engineering, 2015.
[40]	M. Handley, SDP: Session Description Protocol, IETF, 2006.
[48]	A. Baraldi, (2012) ‘Operational Automatic Remote Sensing Image Understanding Systems: Beyond Geographic Object-Based and Object-Oriented Image Analysis (GEOBIA/GEOOIA). Part 2: Novel system Architecture, Information/Knowledge Representation, Algorithm Design and Implementation’ in Remote Sensing Vol 4, issue 9, pp. 2768-2817
[49]	A. Dennis, et al (2015), ‘Systems Analysis and Design: An Object-Oriented Approach with UML’
[50]	L. Liu, (2020) ‘Requirements Modeling and Coding:An Object-Oriented Approach’
[56]	María Emilia Cambronero, et al., 'A Comparative Study between WSCI, WS-CDL, and OWL-S', IEEE International Conference on e-Business Engineering, 2009, Volume: 1, Pages: 377-382.
[57]	OASIS Standards, Web Service Atomic Transaction Specification, version 1.2, 2009, available at http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.pdf
[58]	OASIS Standards, Web Service Business Activity, version 1.1, 2007, available at http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-os.pdf
[59]	OASIS Standards, Web Service Business Process Execution Language, Version 2.0, 2007, available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
[63]	The Open Group, “Open Universal Domain Description Language (Open UDDL),” 2019, [Online]. Available: https://www.opengroup.org/face/docsandtools.

1 Introduction

The MOD's PYRAMID programme introduces a paradigm shift to the current method of avionic systems design and procurement, aiming to make the next generation of air systems affordable, capable and adaptable by the adoption of an open architecture approach and systematic software reuse.

1.1 Scope

The PYRAMID Deployment Guide document outlines how the system and software design processes and strategies used to develop air system software should be adapted when using the PRA, and how PRA artefacts could be implemented by an Exploiting Programme such that the KURs can be realised.

For the purposes of this document, "deployment" and "system" are terms generally used interchangeably to refer to a system implementation based on the PRA.

Note, the lifecycle outlined in this document may differ from that used in a specific deployment activity. This document does not mandate any particular process and it is down to PYRAMID Exploiters to decide which of the considerations outlined would best suit their own processes.

1.2 Purpose

This document provides a guide to engineers about what to consider when developing a PRA based system and how this may affect a more traditional development process.

1.3 Document Structure

Future air systems are expected to be increasingly complex, and this document takes a "separation of concerns" approach to address this. Instead of breaking down the problem space into a hierarchy of ever decreasing pieces, it breaks the problem space down into two stages:

- The functional or platform independent stage of the design, independent of the Exploiting Platform's infrastructure, described by the centre section of Figure 1.
    - Platform Independent System Design provides guidance on how the PYRAMID Reference Architecture (PRA) can be used to assist in developing a Platform Independent Model (PIM) and component specifications.
    - Platform Independent Component Design provides guidance on how the PRA can assist in further component maturation by considering component behaviour based on the component specification.
- The non-functional or platform specific stage of design, usually dependent on the Exploiting Platform's infrastructure and environment, is described by the lower section of Figure 1.
    - Platform Specific System Development provides guidance on how the PRA can assist in the platform design decisions, and in dealing with impacts on the components that arise upon identification of the detailed infrastructure of the Exploiting Platform.
    - Platform Specific Component Development provides guidance on how the PRA can assist in further component development for the identified Exploiting Platform's infrastructure.

The document further distinguishes between what is required from the provider of a component specification, and the component implementation. This, combined with an iterative design process, allows these concerns to be considered independently even though they may impact on each other in some ways.

The sections in this document correspond to a potential deployment lifecycle, where a group of components are taken from the abstract level covered in the PRA through to executing code as part of a final system, as summarised in Figure 1.

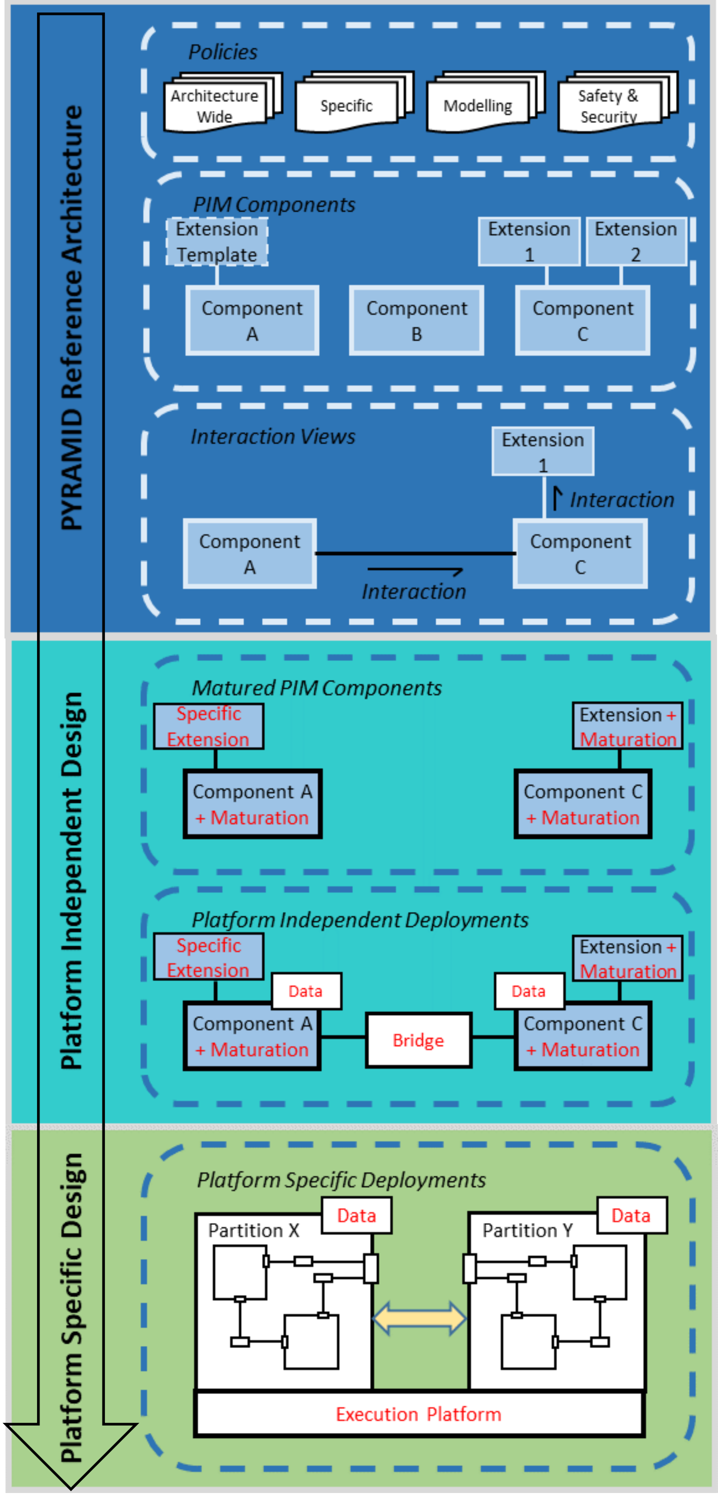


Figure 1: Deployment Lifecycle Diagram

This document has the following sections:

1.3.1 Platform Independent System Design

This section provides guidance on the functional analysis and design of a group of components (or whole system), independent of technology or infrastructure detail, with reference to the PRA.

1.3.2 Platform Independent Component Design

This section provides guidance on the functional analysis and maturation of a component, independent of technology or infrastructure detail, with reference to the PRA.

1.3.3 Platform Specific System Development

This section provides guidance on the analysis and development of a system (or group of components) for an Exploiting Platform's infrastructure, with reference to the PRA.

1.3.4 Platform Specific Component Development

This section provides guidance on the analysis and development of a component for an Exploiting Platform's infrastructure, with reference to the PRA.

1.4 How to Read This Document

Those involved in development of a PRA deployment should use the Description Document, Ref. [2], to familiarise themselves with the PRA policies, components, and example component build sets in the form of interaction views (IVs). Many of the architectural policies are referred to throughout the Deployment Guide, and these can be read in Appendix A of the Description Document, Ref. [2].

The following subsections highlight the sections of the guide that are relevant to different engineering roles. The dark blue items are sections of highest relevance, while the light blue items are less relevant but still contain useful guidance.

1.4.1 Guidance for System Architects and Designers

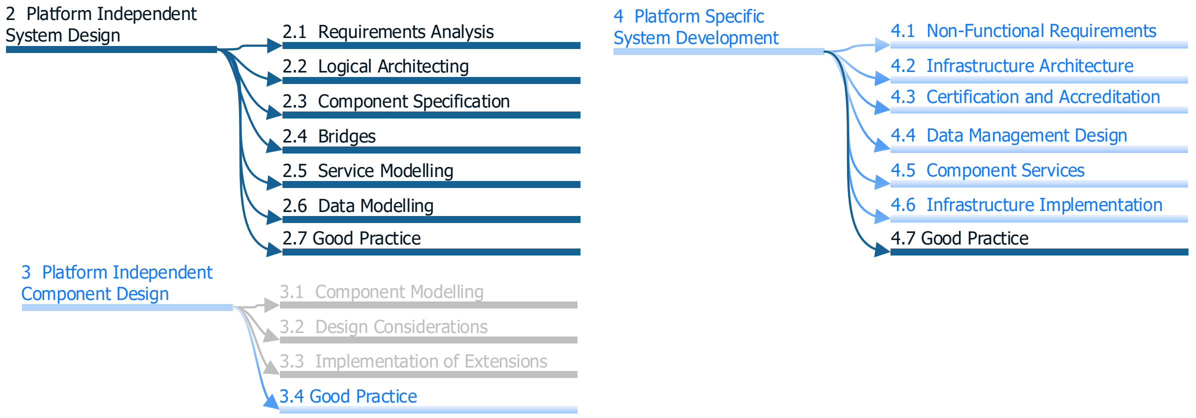

2 Platform Independent System Design
2.1 Requirements Analysis
2.2 Logical Architecting
2.3 Component Specification
2.4 Bridges
2.5 Service Modelling
2.6 Data Modelling
2.7 Good Practice
3 Platform Independent Component Design
3.1 Component Modelling
3.2 Design Considerations
3.3 Implementation of Extensions
3.4 Good Practice
4 Platform Specific System Development
4.1 Non-Functional Requirements
4.2 Infrastructure Architecture
4.3 Certification and Accreditation
4.4 Data Management Design
4.5 Component Services
4.6 Infrastructure Implementation
4.7 Good Practice

Figure 2: Main Sections of Interest for System Architects and Designers

The starting point for those with the role of designing a deployment is section 2. Platform Independent System Design outlines the overarching functional considerations for the deployment and drives out the requirements for the individual components. Some of the technology dependent approaches covered in section 4 will also be of interest.

1.4.2 Guidance for Component Designers

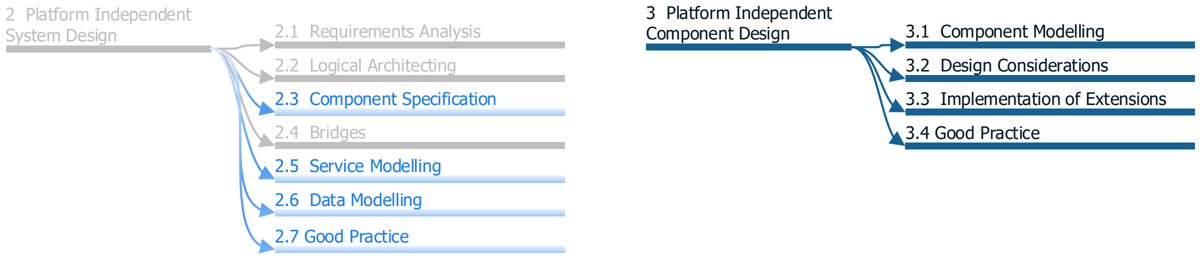

2 Platform Independent System Design
2.1 Requirements Analysis
2.2 Logical Architecting
2.3 Component Specification
2.4 Bridges
2.5 Service Modelling
2.6 Data Modelling
2.7 Good Practice
3 Platform Independent Component Design
3.1 Component Modelling
3.2 Design Considerations
3.3 Implementation of Extensions
3.4 Good Practice

Figure 3: Main Sections of Interest for Component Designers

The starting point for a designer of a component using the PRA is the component specification design, see sections 2.3, 2.5 and 2.6.

Section 3 describes the bulk of the component design work, which can be progressed once the service interfaces have been designed. Platform Independent Component Design outlines the overarching functional considerations for components and defines the work that is expected to be done during component design.

1.4.3 Guidance for Component Developers

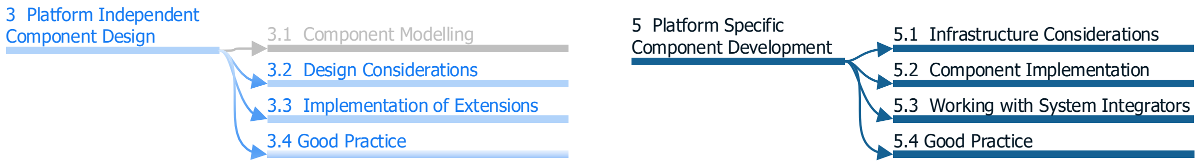

3 Platform Independent Component Design
3.1 Component Modelling
3.2 Design Considerations
3.3 Implementation of Extensions
3.4 Good Practice
5 Platform Specific Component Development
5.1 Infrastructure Considerations
5.2 Component Implementation
5.3 Working with System Integrators
5.4 Good Practice

Figure 4: Main Sections of Interest for Component Developers

The starting point for a developer of a component using the PRA is section 5. Platform Specific Component Development outlines the steps to turn the functional design developed in section 3 into components that will work on the target execution platform.

1.4.4 Guidance for System Integrators

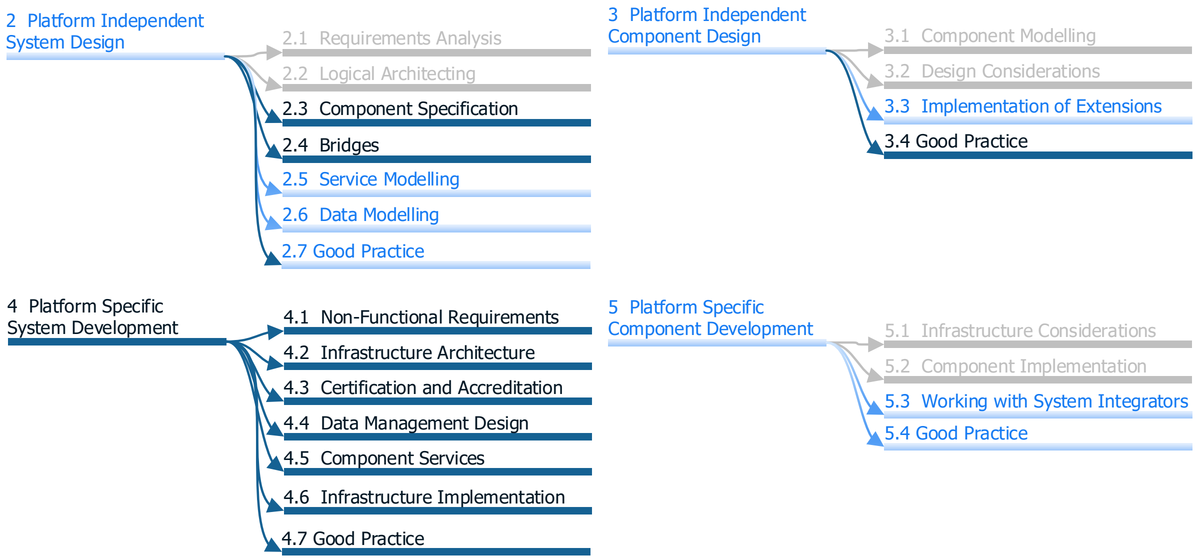

2 Platform Independent System Design
2.1 Requirements Analysis
2.2 Logical Architecting
2.3 Component Specification
2.4 Bridges
2.5 Service Modelling
2.6 Data Modelling
2.7 Good Practice
3 Platform Independent Component Design
3.1 Component Modelling
3.2 Design Considerations
3.3 Implementation of Extensions
3.4 Good Practice
4 Platform Specific System Development
4.1 Non-Functional Requirements
4.2 Infrastructure Architecture
4.3 Certification and Accreditation
4.4 Data Management Design
4.5 Component Services
4.6 Infrastructure Implementation
4.7 Good Practice
5 Platform Specific Component Development
5.1 Infrastructure Considerations
5.2 Component Implementation
5.3 Working with System Integrators
5.4 Good Practice

Figure 5: Main Sections of Interest for System Integrators

Those whose role includes integrating the deployment are likely to be interested to some extent in all the work that is to be undertaken. This role may be covered by a party acting as both designer and integrator, although it may be done as a distinct role separate to the System Architect. Earlier integrator involvement will lead to a smoother integration and acceptance, particularly if the integration authority is different to the system design authority, and this should especially be considered for work described in sections 2.3, 2.4 and 4.3.

2 Platform Independent System Design

This section covers the platform independent system design. *Platform independent* in this context refers to consideration of what functionality is required, with independence from the execution platform and all things that relate to where, when and how the functionality is to be achieved; those are covered in sections 4 and 5. The process, starting with stakeholder requirements, is summarised in Figure 6. Note that this figure does not cover reuse of previously developed components.

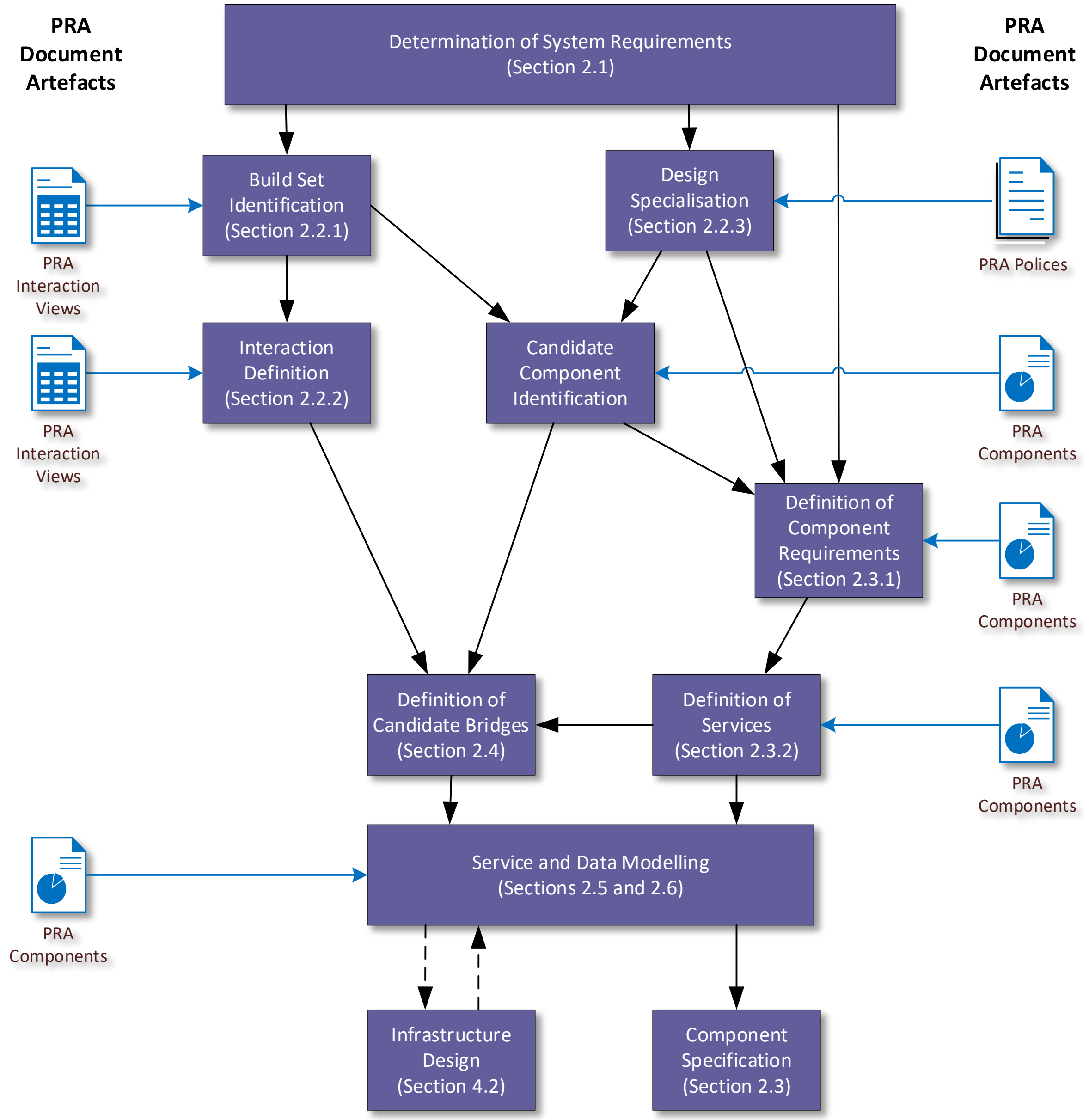

PRA Document Artefacts
Determination of System Requirements (Section 2.1)
PRA Document Artefacts
Build Set Identification (Section 2.2.1)
Design Specialisation (Section 2.2.3)
PRA Interaction Views
PRA Polices
Interaction Definition (Section 2.2.2)
Candidate Component Identification
PRA Interaction Views
PRA Components
Definition of Component Requirements (Section 2.3.1)
PRA Components
Definition of Candidate Bridges (Section 2.4)
Definition of Services (Section 2.3.2)
PRA Components
Service and Data Modelling (Sections 2.5 and 2.6)
PRA Components
Infrastructure Design (Section 4.2)
Component Specification (Section 2.3)

Figure 6: Platform Independent Process

Note that it is expected that the design process will be iterative, with developments in the platform specific modelling informing the logical design; the separation is about considering the functional and non-functional aspects independently.

2.1 Requirements Analysis

The platform independent (functional) requirements applicable to the system need to be captured. The derivation of the system functional requirements from both customer requirements and requirements to interface with equipment needs to be achieved by a normal system engineering process.

2.2 Logical Architecting

Each PRA component is designed to be solely responsible for any behaviour relating to its role, enabling a high level of cohesion, a measure of the component's functional strength. This high level of cohesion in components is an important principle, not only to minimise the impact of changes on the system but also to ensure no component contains information relating to other components. Cohesion requires that components support the architecture in performing common functionality, by for example: providing their own logging information, and reporting their state information to support monitoring (see the Capability Assessment and Health Management policies). It also means the components cover different operational contexts, for example training and simulation, maintenance support, Mission Planning and post mission analysis.

The PRA is applicable to multiple mission scenarios, meaning that it is very broad in scope. Therefore, to develop a deployment based on the PRA a process of specialising the design for specific deployment environments must be undertaken, while still maintaining independence from the execution platform. The rest of section 2.2 provides guidance on the process of logical architecting, which applies this specialisation.

The PRA does not provide an exploiter with a system architecture. To utilise the roles, responsibilities and other information provided in the PRA, it is considered beneficial to have a schematic phase for a deployment. This could consist of features, functionality and hazards "roughly" laid out across the deployment components (including safety and security aspects), comprising an initial logical design. An overall schematic and initial functional design helps reduce risk of costly system redesign later in the design lifecycle.

2.2.1 Build Set Identification

Having identified the functional requirements, the next step is to identify the PRA components that will satisfy those requirements by mapping the derived functional requirements from the requirement analysis work to component responsibilities. Future systems will need to be changed more quickly, more cost effectively and more frequently. Consistent component definitions across systems is core to reuse at all levels (design, functional, and software), making mapping system requirements to the components a fundamental first step.

Artefacts to assist with build set identification are included within the PRA Model:

- Interaction views (IVs), which show how higher level functions may be achieved using a combination of components. The IVs are included in Appendix C of Ref. [2].
- Trace links, which indicate the component responsibilities that interaction events in the IVs correspond to.
- Architectural policies, which discuss architecture wide provisions as well as more specific functional areas. The policies are included in Appendix A of Ref. [2].
- Semantics diagrams within each PRA component definition, which contain entity relationships that identify the scope of the component and therefore indicate whether it should be included in the build set. These diagrams are included in Appendix B of Ref. [2].
- Service Definitions, which define a basic set of services for each PRA component. These definitions can assist in determining where build set components will interact. These definitions are included in Appendix B of Ref. [2].

The identified components need to be captured into a PIM build set. A PIM build set is effectively a parts list that specifies the selected components that make up a specific deployment of the PRA, and their interconnections. An example diagram for a build set of components used in aerial refuelling is shown in Figure 7. Typically a build set is documented in a composite or package UML structural diagram; multiple tiers of build set may be required to capture a complete deployment. The PRA IVs are example use cases (which also include interactions, see Section 2.2.2) that can be consulted when identifying relevant components for related build sets. More details on how to develop a build set are provided in Ref. [9] and [13].

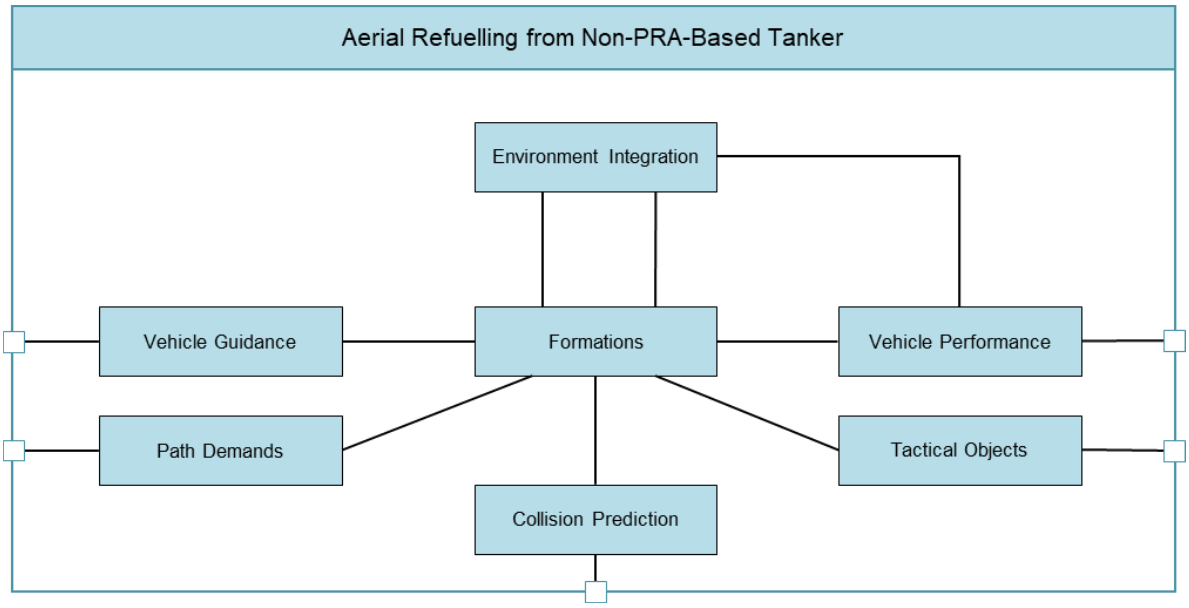


Figure 7: Example Build Set Diagram

2.2.2 Interaction Definition

There are many different ways that the components can be combined. The interaction views (IVs) provide examples of how components can be used together to meet a higher level design goal.

As part of the platform independent design process, the interactions between different components in a PIM build set need to be elaborated. Initially these are likely to be based around simple component-to-component interactions. Multiple techniques for identifying the initial deployment-specific interactions can be used, e.g. usage analysis. At this stage it can be beneficial to produce use cases based on what is required from the deployment, or a particular part of it. Multiple use cases can be investigated within larger use cases, and the relationships between them in terms of how they use each other can be determined, as well as the relationships with external actors.

Component interactions for a build set are typically captured in sequence or communication behavioural UML diagrams. Figure 8 shows a communication diagram that defines interactions for the aerial refuelling example (introduced in Figure 7). The communication diagrams used in the PRA IVs (Appendix C of Ref. [2]) show event interactions, but data flow interactions could potentially be used, as in Figure 8.

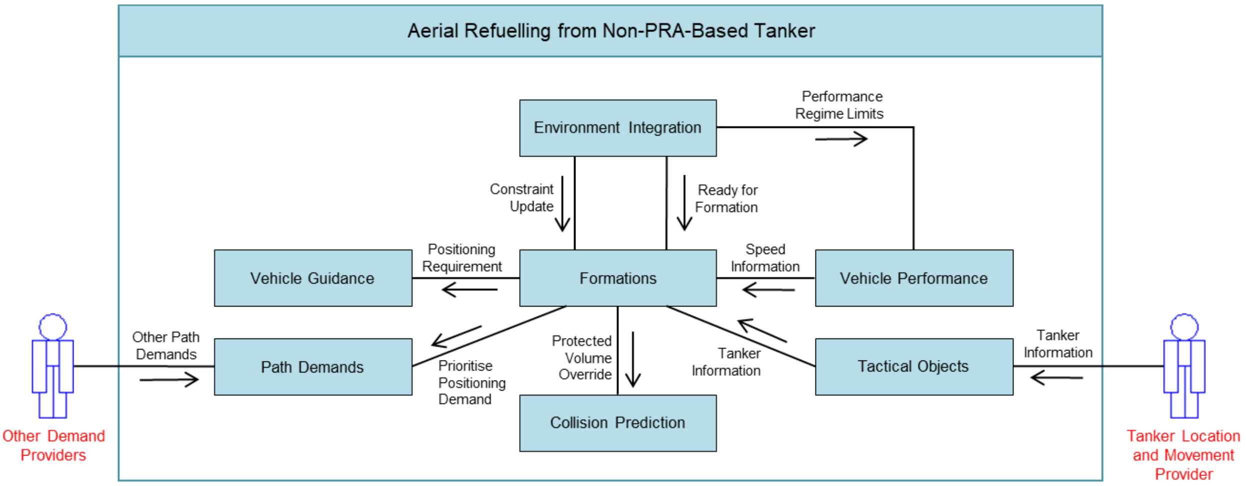


Figure 8: Example Representations of Initial Interactions

The PRA IVs show example interactions between components, which are satisfied by their services. These interactions can be matured into fully defined service interfaces, as described in section 2.3. For initial iterations, keeping components and services generic will help promote reusability in future iterations. Further guidance on component design specialisation is covered in section 2.2.3.

2.2.3 Design Specialisation

Any design process is not going to be a single pass, as the design process for non-functional requirements will impact the functional design. This means that the design needs to evolve so that the PIM, while still independent of the execution platform, continues to be tailored for a particular deployment as the infrastructure and non-functional requirements are matured. To perform logical architecting of the system, component design specialisation should be considered. The architectural policies need to be considered during specialisation, particularly the Component Extensions and Data Driving policies. The following sub-sections provide additional recommendations on tailoring of a design using the PRA to assist in meeting specific stakeholder requirements.

Note: these sections are not meant as an exhaustive list of tasks; additional tasks may be necessary, and those applicable will depend on the methodology being followed.

2.2.3.1 Autonomy and System Constraints

Air systems of the future are likely to include many automated decision making functions. The ability to include multi-level decision making is at the core of the PRA design.

- The **Control Architecture** policy describes a control architecture that has been embodied in the PRA, enabling the architect of an Exploiting Platform to implement system-wide control.
- The **Autonomy** policy provides recommendations on capturing the level of autonomy that is deemed appropriate.
- The **Constraint Management** policy provides recommendations on handling constraints associated with rules and limits, etc.
- The **Capability Assessment** policy describes how changes in component and wider system capability should be detected, and use of capability information in determining response to the changes.

The level of autonomy that is desired in the system, and the handling of system constraints, should be considered as part of the platform independent system design. A principle of the Control Architecture policy is that decisions are made by the component that has the appropriate level of information (e.g. detailed low level information) and subject matter knowledge to make the decision.

Note that although some components have oversight roles within the control architecture, they are still agnostic of where the tasks are carried out (i.e. they are still unaware of what other components are present in the build set). The control architecture is not a solution for bridging the semantic gap between components (see the Component Connections policy for this).

2.2.3.2 Component Scaling

For a given deployment, it may be desirable to specialise the functionality of components for resourcing, temporal or spatial reasons (e.g. offline planning or ground support). Each component can be scaled into distinct component variants containing a subset of the component’s responsibilities; each variant can then be tailored for specific resource profiles or operational contexts.

When scaling a component it is important to avoid producing high coupling between the component variants. Given that variants will share the subject-matter knowledge of their original component, this needs some thought; meaningful and comprehensive boundaries between the variants are required in order to ensure the components can operate without each other, or so that they can co-operate efficiently.

Component variants may be used to distribute a component’s functionality to address resourcing, security or safety concerns. If this is the reason for the variants, they are likely to need synchronising with each other. These considerations are covered in section 2.2.3.3 and 2.2.3.4. Additional guidance on distributing components by context is provided by M. Page-Jones who proposes several key strategies for developing coherent components; see Ref. [24] [48] [49] [50].

2.2.3.3 Multiple Instances of Components

Whereas scaling a component can enable specialisation of its overall functionality for aspects like different operational situations or settings (see Section 2.2.3.2), more specific details can be specialised using multiple actual instantiations of the same component (or variant thereof). This may be done for numerous reasons, for example to:

- Distribute or compartmentalise behaviour for resourcing, security and/or safety reasons. Security and safety concerns should be considered together, see section 4.3.
- Instantiate a data-driven generic component multiple times for specific purposes.

The communications between multiple instances of components need to be kept to a minimum. The service interfaces must be as carefully considered as those between different component variants (the distributed parts) – this will not be achieved by treating it as a single component.

The following architectural policies offer recommendations on distributed working:

- The **Multi-Vehicle Coordination** policy contains guidance on the orchestration of tasking over a system distributed between multiple geographic locations;
- The **Use of Communication** policy contains guidance on how the data communications work together to provide a managed multi-node solution;
- The **Operational Support** policy contains guidance on how instances of components can be used in the planning and operational usage of the system.

Where instances of the same component co-exist, they are most likely to be changed by *configuration* with particular requirements incorporated specific to the Exploiting Programme, enabling the component to be more accurately tailored to its intended role. An example might be multiple instances of a resource component with each instance configured to operate as a specific type of resource (e.g. a sensor type), see the Data Driving policy.

Note that using multiple instantiations of a generic component for a particular purpose is a choice; a single parent instance could instead be used as the final arbiter of its extension components. For example, a single instance of a resource component could delegate behaviour for different resource types to the extension components responsible for behaviour specific to that resource type. For information on component extensibility, see section 3.3 and the Component Extensions policy.

2.2.3.4 Data Synchronisation

The PRA components are designed to be highly cohesive, partially to minimise the amount of data sharing. When a component is scaled or distributed, interactions need to be created for the required synchronisation between the different instances of the same component. This will often occur naturally as part of the expected interactions with other services, and does not require any additional dedicated design. These new interactions are important because they may not have been covered in any previous interaction views.

Synchronisation needs to be analysed; initially it is sufficient to consider existing interaction definitions to cover synchronisation. Section 2.6 suggests how data that may require synchronisation can be analysed, so that it can be better understood.

2.3 Component Specification

Based on the interaction definition identified as part of logical architecting, it should now be possible to better refine what is required of each component. It is recommended that component specifications are iteratively refined in a collaborative manner with the Component Designer. Although the Component Designer does not have to be involved when the specification is first defined, they should be involved as early as possible.

The component specification defines the purpose of a component in terms of what it provides and what it needs from the system. This allows for parallel work to be carried out on all the components, and helps determine whether components are going to meet the system needs. When specifying a component for a deployment, any specified terms that are particular to the deployment should be aligned with or related to the language of the PRA definitions on which they are based. For example, when specifying a specialised interface based on a PRA service definition, the relationship between specialised attributes and any equivalent attributes in the PRA definition should be stated (if they are not going to be aligned). This will help maintain traceability and understanding.

Depending on the level of reuse being undertaken in the deployment, a component specification may already exist. Reuse of components should be maximised at this stage so that software and artefacts with pre-existing functionality are not developed unnecessarily. Component specification typically covers three main areas:

- **Functional Requirements** – requirements that have been derived for the component during system analysis. See section 2.3.1.
- **Services** – define what is provided by the component and what it requires in return, see section 2.3.2.
- **Service Dependencies** – define the relationship between services the component provides and consumes, see section 2.3.3.

For compatibility and consistency, component specification functional requirements and service definitions should specify the detail for how the component will be required to support the necessary PRA policies. This should be specified while considering both the policy's direct implications for the component and also the effect on the wider deployment. For example, the way that different requirements should be communicated through the component's services will support the Dependency Management architectural policy. The policies are included in Appendix A of Ref. [2].

As the component specification is refined and the design better understood, safety, security and autonomy related dependencies can be captured. Section 3.2.5 discusses further detailing of these dependencies to enable later component implementation, particularly with respect to safety. Section 4.3 discusses addressing of non-functional requirements at the Platform Specific Model (PSM) stage.

2.3.1 Functional Requirements

Component requirements need not be textual. One of the key goals of model based design is to provide a more automated way of progressing from high level requirements to running software. Ideally, to support this the requirements should be captured in a machine interpretable way, with the component supplier and System Integrator having an agreed approach to electronic exchange of requirements, traceability and verification evidence.

2.3.2 Services

Services are at the core of the component specification; a service may be as simple as providing a piece of information or as complex as generating a complete flight path. There are many approaches to defining the service interface to components; while the PRA does not mandate any particular methodology for service design the PRA has been designed with support for the following modelling methodologies in mind:

- A service modelling approach as detailed in section 2.5.
- A data modelling approach as detailed in section 2.6.
- A domain modelling approach as detailed in Ref. [13].

Multiple approaches can be used at once, but often a dominant method is required across a deployment to avoid multiple definitions of the same service; for more on this see Ref. [8]. Each PRA component definition (included in Appendix B of Ref. [2]) provides basic services which can be used as the basis for the services in a component specification.

2.3.3 Service Dependencies

A service dependency explains how the services a component provides depend on other provided or consumed services. The dependencies between a component's service activities can be represented in activity diagrams, of which a generalised version is shown in Figure 9. Aspects such as events and signals can be included in an activity diagram to provide the desired level of detail.

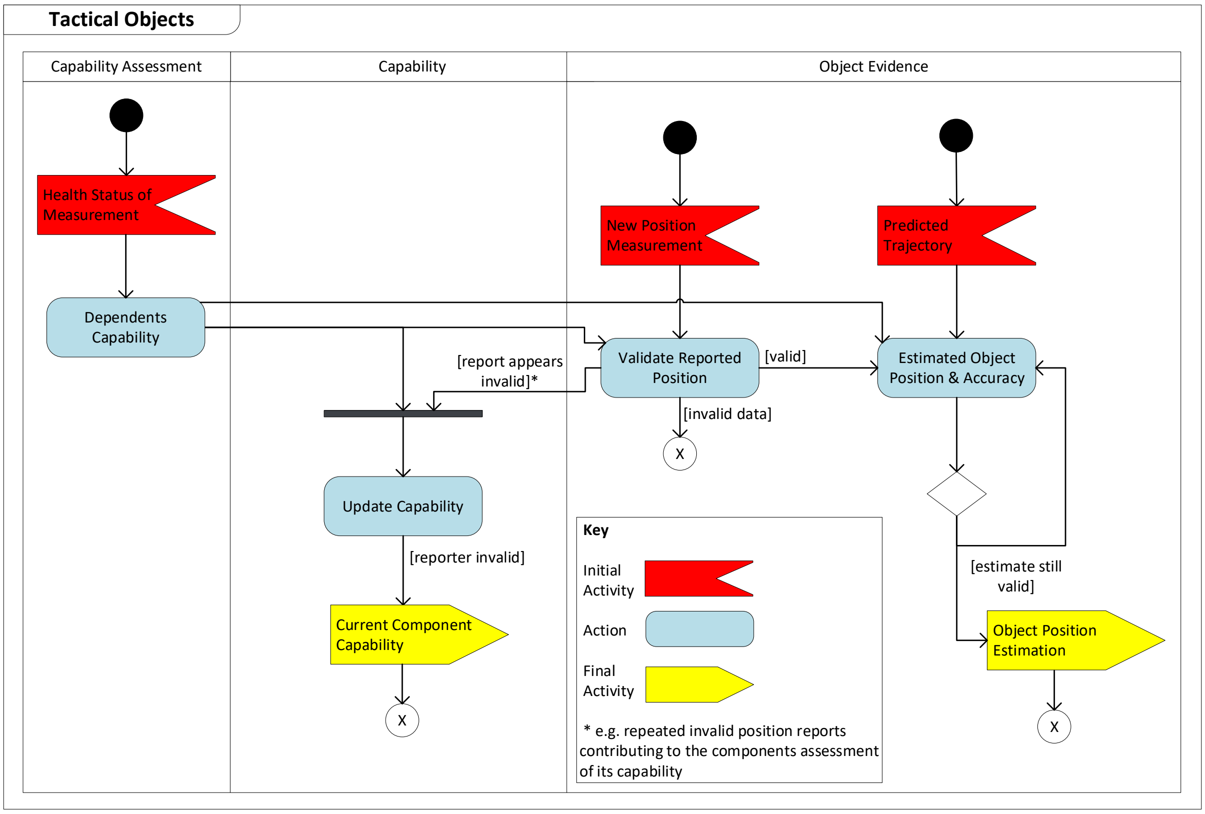

Tactical Objects
Capability Assessment
Capability
Object Evidence
Health Status of Measurement
Dependents Capability
[report appears invalid]*
Update Capability
[reporter invalid]
Current Component Capability
X
New Position Measurement
Validate Reported Position
[valid]
[invalid data]
X
Predicted Trajectory
Estimated Object Position & Accuracy
[estimate still valid]
Object Position Estimation
X
Key
Initial Activity
Action
Final Activity
* e.g. repeated invalid position reports contributing to the components assessment of its capability

Figure 9: Service Dependency Diagram

As the components are refined during a deployment, these diagrams should be modified and matured. The PRA includes a service dependency activity diagram in each component definition, and these can be used to inform a component's service dependencies for a deployment.

A service dependency is the part of the component specification that has to be the most co-operative in its development, as it is used to tell a System Integrator what has to be provided to the component in order to get the service that is requested.

2.4 Bridges

The need for information to be exchanged, and to overcome semantic gaps between components, is met by passing information via special inter-component connections known as bridges.

Part of the design process is to identify these bridges to make sure that the gap being crossed is not too wide, and to ensure the complexity that needs to be reasoned about in the architecture is not being hidden in the bridges. Approaches to specifying this behaviour are provided in the Component Connections and Use of Communications policies. Additionally, there are examples of bridges within the standardised connection patterns provided in Appendix A of this document.

Within the PIM, model partitioning and timing are not considered; the model 'assumes' that all interactions are instantaneous and allowed. The platform specific modelling described in later sections will address non-functional constraints such as partitioning and timing; where this changes the logical behaviour, an iterative approach can be undertaken to capture this in the platform independent model if required.

Section 4.7.6 describes the use of component subject matter entity counterparts later in the design process to enable bridges to be implemented in a code efficient way.

For more information on bridges see the Component Connections policy.

2.5 Service Modelling

Service modelling methodologies such as Service Oriented Architecture (SOA) start from the assumption that the component specification can be fully defined by its services alone. This means that, unlike data modelling (see section 2.6), when defining the component the behavioural aspects such as operations are considered but the associations between the subject matter entities are not. These methods complement the approach used to create the components defined in the PRA, as the key service characteristics are very similar. For more information on SOA see Ref. [11] and [12].

A service modelling approach is often iterative and subdivides the platform independent model into separate stages of development, providing further separation of concerns. Example separation is shown in Figure 10. This can represent the evolution of a single model, however in an auto-translation environment this will often be multiple models with approximately the first 80% of the next layer auto-generated from the model above; see section 4.6.2 for more on auto-translation.

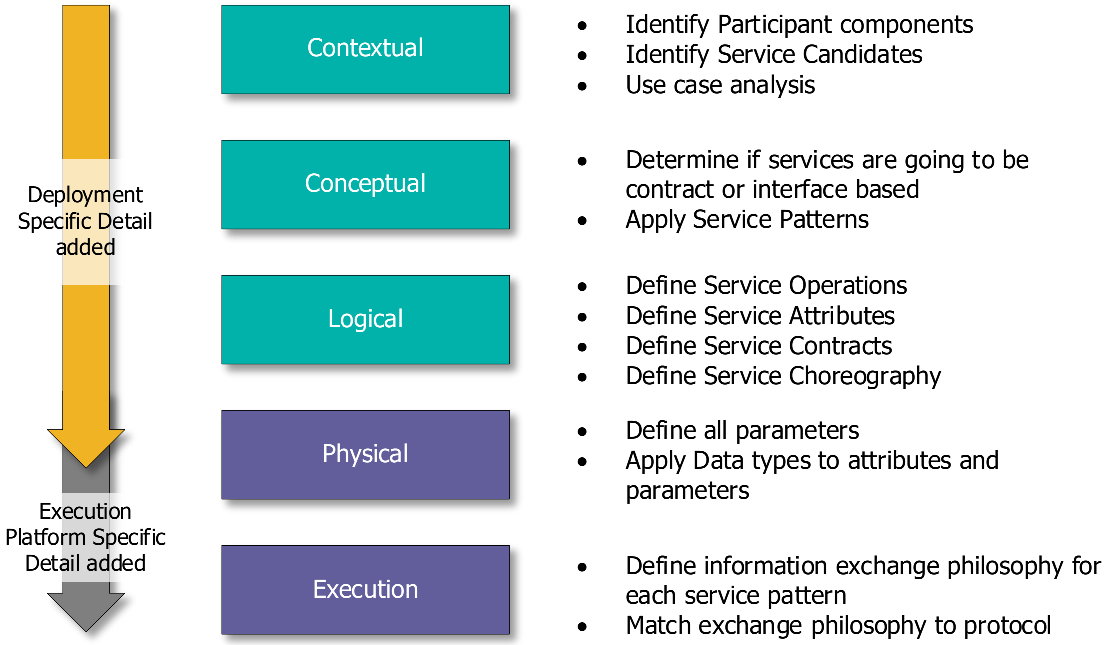

Contextual
Identify Participant components
Identify Service Candidates
Use case analysis
Conceptual
Determine if services are going to be contract or interface based
Apply Service Patterns
Deployment Specific Detail added
Logical
Define Service Operations
Define Service Attributes
Define Service Contracts
Define Service Choreography
Physical
Define all parameters
Apply Data types to attributes and parameters
Execution Platform Specific Detail added
Execution
Define information exchange philosophy for each service pattern
Match exchange philosophy to protocol

Figure 10: Example Set of Service Modelling Stages

Note that the set of stages shown in Figure 10 is a progression through both PIM and PSM levels, without a strict boundary between them.

2.5.1 Model Artefacts

Good service definition is essential for efficient integration of a deployment. Service modelling provides two types of services at the PIM level:

- **Provided services**, which are services offered by the component;
- **Consumed services**, which are services required by the component from elsewhere in the system.

Typically component specifications will provide the following artefacts related to services:

- **Service interfaces**, which define the functionality provided or consumed by the component. Interfaces may contain service operations and/or attributes that are each related to the overall provided or consumed service.
- **Component centric service contracts** that define which service aspects must be adhered to so that the service can form a relationship in order to provide a capability. For more information see the Component Connections policy.

Service interfaces can be modelled in two ways:

- A **simple service**, where the interface is defined as attributes on a service interface class that can be mapped via a data-only relationship in a similar way as in data modelling, e.g. where the interface is just passing data.
- A **contract service**, where a simple interface cannot fully represent the expected behaviour, e.g. if the interface is requesting an action rather than just passing data.

These two are often combined in the same service definition (e.g. an attribute that is used in a simple interface to report the progress on the task in the contract interface). This eliminates the need for "get/set" operations as part of the platform independent model, allowing for a simpler representation.

A **service pattern** defines a description of a common solution to a recurring problem within a certain context. See the Component Connections policy for more on service patterns. Appendix A of this document contains examples of applicable service patterns in use.

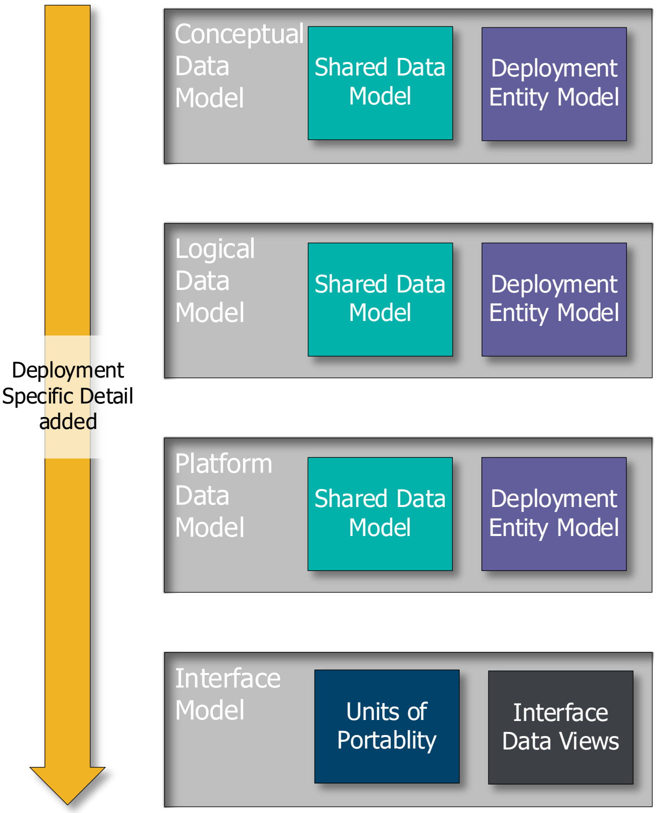
The basic services provided in the PRA (Appendix B of Ref. [2]) should be specialised for the needs of a deployment. A base set of attributes are included on Service Interfaces in the PRA model, and these can be refined or additional attributes can be added as services are developed further. The PRA interaction views (Appendix C of Ref. [2]) provide examples of how components interact, which can help when determining service operations; the interactions can be considered as needs to provide and consume services, which in turn may be realised by more than one service operation.

When defining attributes and operations on the services it is important to remember that the service object only exists for the life of that service being started, so it cannot capture and retain information for it to be called elsewhere. However complex services are allowed where they are set-up or planned via one operation, and the data is retrieved by a different attribute or operation.

2.6 Data Modelling

Data modelling methodologies start from the assumption that the component specification can be fully defined by a component's entities, their attributes, and the relationship between the entities. This means that unlike service modelling (see section 2.5), when defining the component the relationship between subject matter entities are considered, but behaviours and behavioural aspects such as operations are not. The semantics diagrams within the PRA component definitions, despite not being complete class diagrams, give context to the component's entities that can be useful in development of a data model. Data modelling is often done using a bottom up approach where the data provided by a component is already known and is defined as being available to external parties. Data modelling can be used as a stand-alone approach or in combination with service contract definition to help develop a cleaner integration. For data issues relating to component design see section 3.2.

A data modelling approach, like service modelling, is often iterative and subdivides the platform independent model into architectural layers to provide further separation of concerns. Future Airborne Capability Environment (FACE), Ref. [26], for example, separates the model into conceptual, logical and platform (i.e. physical) models; an illustration of the modelling approach is shown in Figure 11. Note that the logical and physical models can often be generated through auto-translation based on a set of predefined rules; see section 4.6.2.



- Define Entities in terms of Observables e.g. Temperature

- Define entities in terms of Measurements e.g. Centigrade
- Define relationship to Conceptual Model
- Define expected data ranges and accuracies

- Define entities in terms of data primitives e.g. integer
- Define relationship to Logical Model

- Define what will be in each interface view
- Define how the deployment entity model will be broken up to support partitioning

Figure 11: Example Set of Data Model Stages

Note that the set of stages shown in Figure 11 is a progression through both PIM and PSM levels, without a strict boundary between them.

2.6.1 Model Artefacts

Across a deployment, a **shared data model** is used that contains the lowest level data type definitions. This allows shared physical measurements and observations so that conversion overheads between deployed programmes are reduced.

The term data model can refer to either a **component information data model** or a **system information data model.**

- Component specifications are defined using component information data models, user concept views and data views, see section 2.6.1.1 for more information.
- The relationships between components are captured in system information data models, see section 2.6.1.2 for more information.

2.6.1.1 Component Information Data Model and Data Views

A Component Specification can provide the following artefacts:

- A **component information data model**, defining the data that is used within a component, and **data views,** defining which parts of the entity data model are exchanged over a particular interface.
- **User concept views**, incorporating customer focused component use cases, therefore only showing the parts of the deployment that the component is directly involved in.

An information model is represented as an entity, relationship and attribute (ERA) diagram, which shows a logical breakdown of the data structure within the component.

The information model can be used to define the component specification, as the understanding of the relationships between the entities is important. Although the full representation of the data available for a component is captured in the information model, often only a subset is required to be shared with other components; the relationships involving data made available outside of the component are modelled as data views. A view is a projection of attributes within the information model, see Figure 12. These are often represented as counterparts, as described in the Component Connections policy.

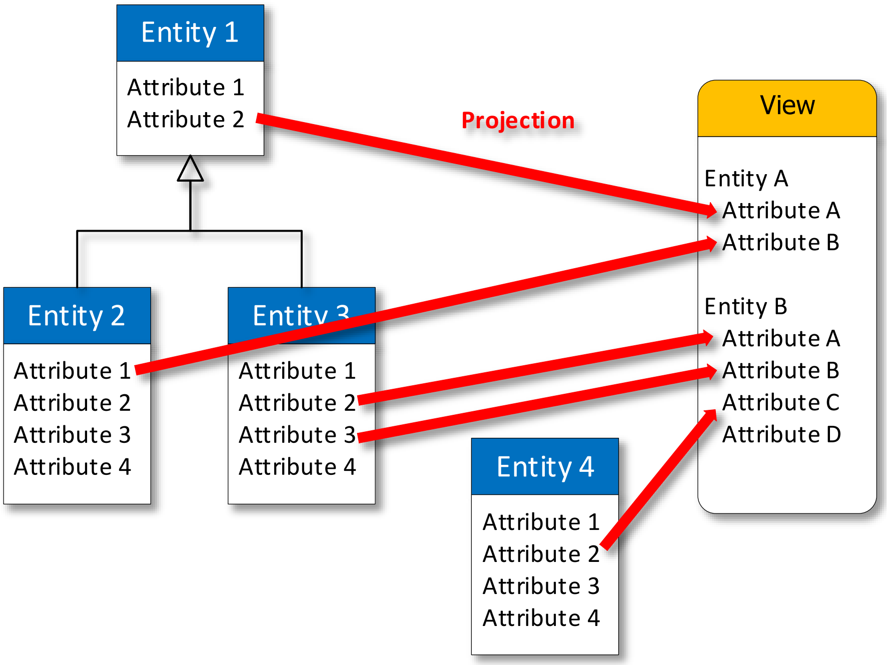


Figure 12: Data Views Contain Projections of the Subject Matter Attributes

System Designers should assess whether an information data model is required, and if so whether it should be a single model or subdivided into relevant sub-models. Where necessary, the management of links and overlaps in multiple data models should also be considered. Only the parts of the information model that are required for the interface specification need to be released to third parties, which is important where Component Developers are concerned with the privacy of their information model.

2.6.1.2 System Information Data Model

A reasonable subdivision within a system data model for a typical programme might be:

- **User concept views**, incorporating customer focused system use cases.
- A **system information data model**, describing the subject matter data groups. The subject matter entities within the PRA can be used as a starting point for the data model.

The information model can be used to:

- Model parts of the system, particularly the data-driven parts of a component.
- Define the semantic data exchanges across multiple platforms operating in a common operating domain.
- Support reuse of existing data types across multiple deployments prior to contract placing.

In the same way that a component information data model is used in combination with data views to define the component's relationship to its interfaces (see section 2.6.1.1), the system data model can be used to capture the relationship between different components. Again a layered approach here can bring reuse advantages, including defining bridge conversion routines.

2.7 Good Practice

This section provides good practice recommendations applicable to typical cases of platform independent system design, including guidance for avoiding common pitfalls.

2.7.1 Do use other standards and models in conjunction with the PRA

The PYRAMID Reference Architecture is not designed to contain all the information that is required to design a fully interoperable end system. Rather it is designed to provide a common component structure and overarching policies which when combined with other architectures and models provide a full definition of a system architecture, see Figure 13.

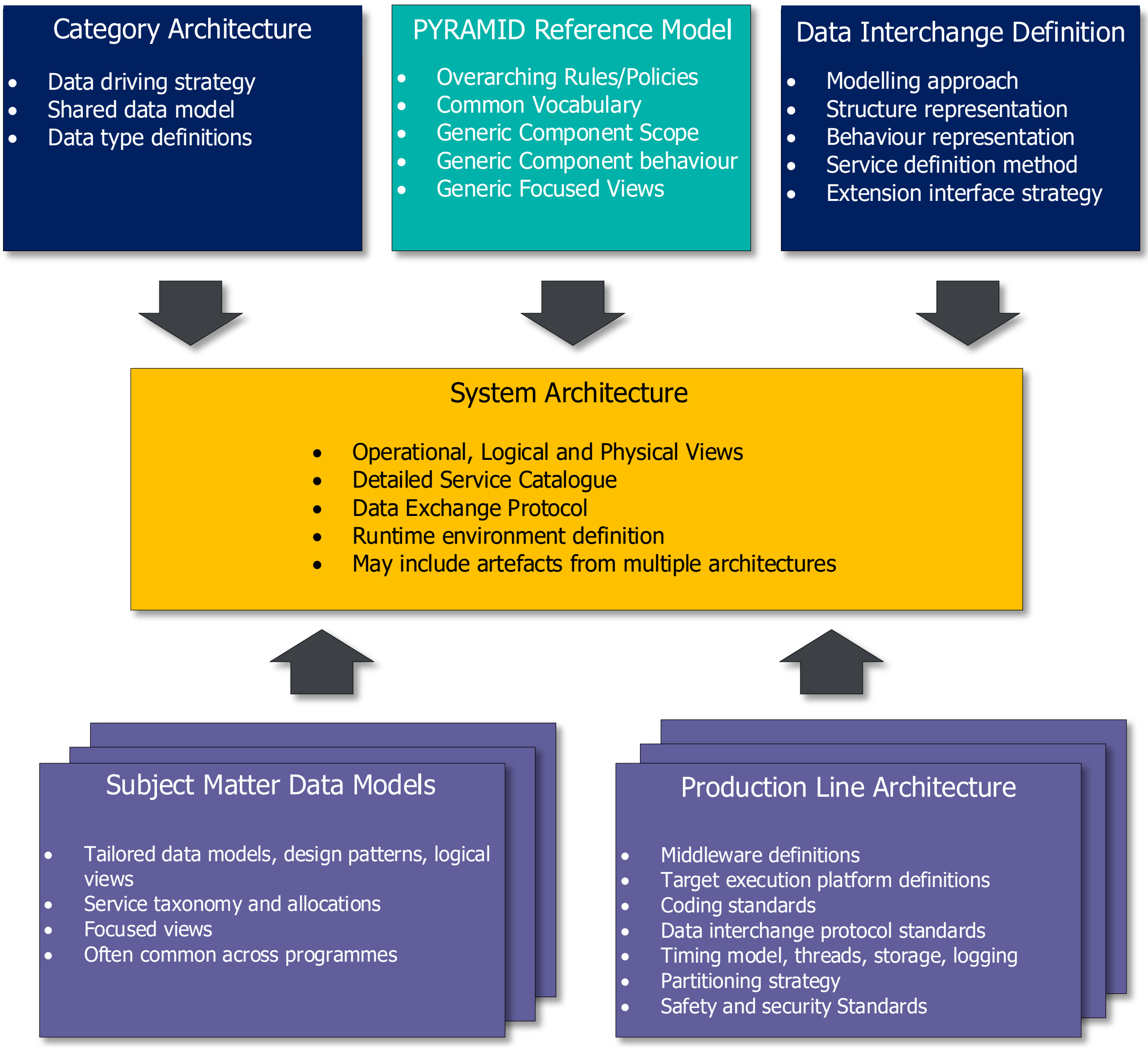

Category Architecture
Data driving strategy
Shared data model
Data type definitions
PYRAMID Reference Model
Overarching Rules/Policies
Common Vocabulary
Generic Component Scope
Generic Component behaviour
Generic Focused Views
Data Interchange Definition
Modelling approach
Structure representation
Behaviour representation
Service definition method
Extension interface strategy
System Architecture
Operational, Logical and Physical Views
Detailed Service Catalogue
Data Exchange Protocol
Runtime environment definition
May include artefacts from multiple architectures
Subject Matter Data Models
Tailored data models, design patterns, logical views
Service taxonomy and allocations
Focused views
Often common across programmes
Production Line Architecture
Middleware definitions
Target execution platform definitions
Coding standards
Data interchange protocol standards
Timing model, threads, storage, logging
Partitioning strategy
Safety and security Standards

Figure 13: Example of Multiple Sources Contributing to a Full System Architecture

The advantage of the mix and match approach is that it allows for end systems to be standardised while still being tailorable; allowing for an instance of a deployment to be Multilateral Interoperability Programme (MIP) compliant and PRA based at the same time. For more on the relationships between different reference architecture types see Ref. [8].

A category architecture typically identifies features that are common across multiple programmes such as canonical models, e.g. a FACE Shared Data Model, Ref. [27]. A programme will need to assess if an information data model is required, and if so whether it should be a single or subdivided data model (e.g. a separate model of data for data driving).

Similarly, the use of subject matter data models allows a common model to be shared between programmes, increasing the compatibility between programmes and also driving down cost. Subject matter models may be defined by the programme for any specific deployment, or subject matter specialisations may be used such as:

- Battlespace Situational awareness e.g. MIP Information Model, Ref. [28];
- Airspace integration e.g. AIXM, Ref. [29];
- UAV Command and Control e.g. OSD UCS, Ref. [30];
- A conceptual model conformant to FACE metamodel, Ref. [27] and [63].

Individual programmes should specify and share their logical and physical layers for detailed enumerates and logical representations, such that interface code could be auto-generated through mapping to the relevant entities in the category architecture conceptual layer.

2.7.2 Do delegate resource conflict decisions as low as possible

A principle of the Control Architecture policy is that decisions are made by the component that has the appropriate level of information and subject matter knowledge to make the decision. Where decision conflicts arise, these should then be made as close to the end of derived requirement chains as possible. This means that components close to the interfaces with resources are encouraged to make the resource level decisions. The higher levels of the control architecture are abstracted away from the resources and are therefore not likely to be able to make the detailed decisions. Components with the broader understanding of the subject only need to be involved if the components that understand certain specific detail can’t find a way to meet the requirements they have been given. See the Dependency Management and Resource Management policies for more information.

2.7.3 Do consider which component is most appropriate for managing certain calculations

When a component requires data to be calculated that is not yet available, it is often the responsibility of that component to instigate the calculation of the data when needed. However, a design decision needs to be made concerning which component should actually perform the calculation (and therefore what data needs to be transferred and to where). Various considerations should be weighed against one another when making these decisions:

1. **Keeping the calculation within the component most concerned with the result.** This has the benefits associated with the calculated data being more easily available where it is needed, such as reduced latency or data transfer overheads if the result contains a large amount of data.
2. **The amount of data needing to be transferred.** Performing the calculation in a component that contributes a significantly larger data set to the calculation will reduce data transfer overheads and potentially latency in producing the result. Note that:
    - Whilst it may be possible to narrow down the amount of data needed for a calculation within a large data set, additional design or computational effort may be required to achieve this efficiency.
    - Thought should be given to whether current technical limitations that restrict large scale data transfer may be reduced in future.

3. **Keeping the calculation in one place.** Where there are common elements or a common theme across calculations, it may be beneficial to perform the calculations in the component that contains or is responsible for the common elements. Whilst common libraries (e.g. mathematical libraries) do enable multiple components to use the same library function without having to redevelop it, there may still be benefits to restricting the number of components that perform similar calculations, such as easier integration of library changes.
4. **Safety and security considerations.** The location of the calculation may need to take account of:
    - The required or desired integrity of the component and/or the hardware that it is intended to be hosted on.
    - The possibility of source or output data corruption, or surveillance during transit, and any additional overheads required to mitigate this.

The significance of point 1 verses point 2 will require assessment to determine where the greatest performance or design effort benefits lie. For example, a conflict between a proposed route and the terrain would be of most concern to the Routes component rather than the Geography component. However, performing the calculation in the Routes component would likely require large amounts of terrain data to be transferred to the Routes component, rather than a much smaller amount of route data (e.g. represented as a line or simple volume) being transferred to the Geography component. Furthermore, there will likely be other cases where lines or volumes, representing things in the subject matter of other components, need to be compared with terrain to determine if they intersect. Therefore, performing the calculation in the Geography component may be the best solution from both a performance and design effort perspective.

2.7.4 Do be aware of emergent properties and behaviour

Emergence occurs where a group of components have properties and behaviour when interacting as a group that no individual component has on its own.

There are two main problems related to emergence that should be considered. Firstly, desired emergent behaviour is not immediately apparent in the component set. Secondly, undesirable behaviour may result from the components being unaware of the behaviour of each other.

Many examples of how to model desired emergent behaviour are covered in the interaction views. Emergent properties resulting from emergent behaviour can often cause confusion, as it may seem that they should be properties within an individual component. These properties can often be identified in that they can only be described clearly at an abstract level, covering multiple subject domains: one such concept is the context of a mission. Mission context information on, for example, an area of operation may be represented by a set of related data from different components, and may not appear with an 'area of operation' label in any individual components. However it may well appear in a more abstract component that understands the subject broadly, but with certain details distributed between other components that understand those details specifically.

Undesirable emergent behaviour is a problem with any system which is an integration of its constituent parts. However, this is a greater problem with a design for which separation of concerns is the underlying philosophy due to the deliberate 'independence' of components and emphasis on autonomous behaviour. This may lead to issues such as undesirable 'loops' of behaviour, or over processing (e.g. filtering) data in different places which can result in degraded quality and unacceptably extrapolated data. These issues can generally be mitigated through being careful with and understanding the implications of how components interact during design. Careful analysis of alternative scenarios and dynamic constraints can be helpful.

2.7.5 Do not create a stove-piped design

One of the hallmarks of a legacy style functional decomposition is the *stove-pipes* it creates, where lower level resources are *under the control* of a dedicated higher level component, with all access going through that high level component (see Figure 14). This leads to a high degree of coupling, in turn leading to code duplication and increased design change latency; clearly this is to be avoided in new systems.

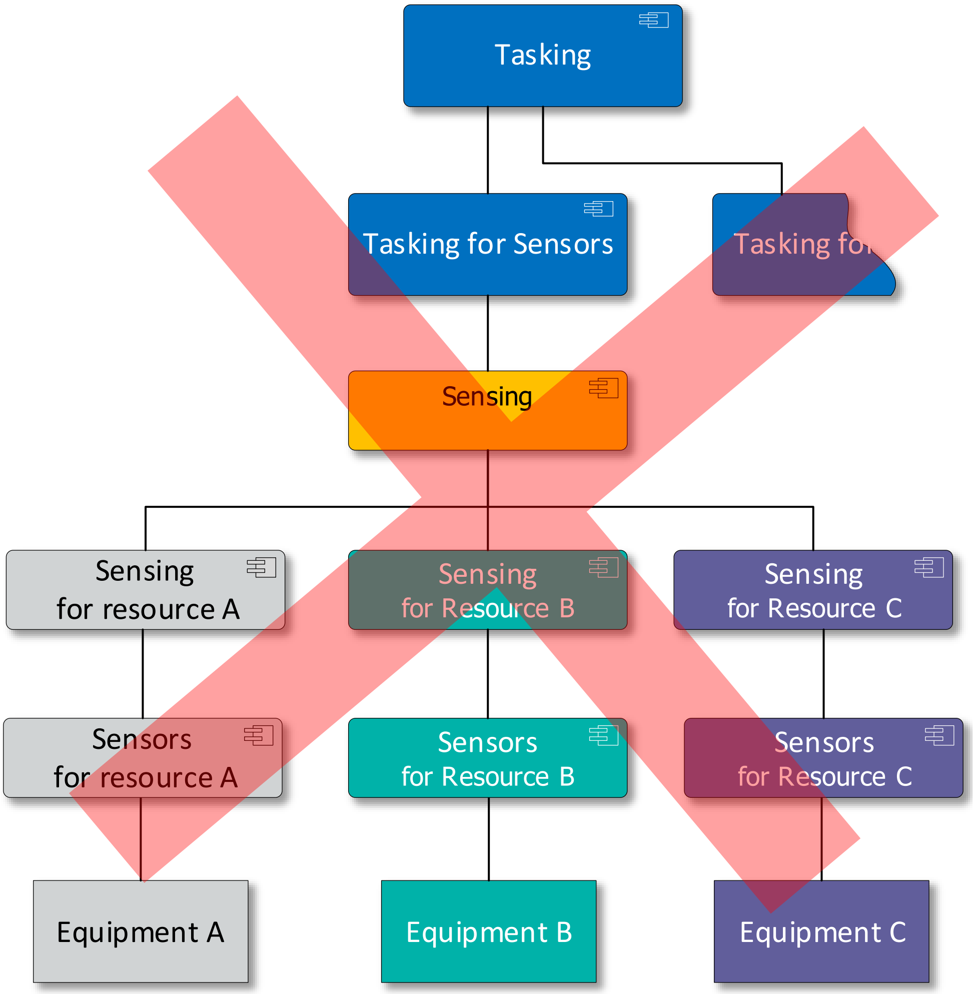


Figure 14: Do Not Use Multiple Copies of the Same Component in Stove-Pipes

If a specialisation of a component's role is needed to support some of the behaviour in new equipment, this should where possible be accommodated using extensions rather than variants. The impact of this specialisation will decrease as you go up the abstraction layers in the control architecture: if you need specialisations in a long chain it suggests that abstraction is not taking place, as shown in Figure 14. Decisions should be made by a component to which all the required information detail is available, and detailed information should not be available at the higher abstraction levels. For instance, it is reasonable for a Tasks component to decide that to meet intelligence objectives a sensing action is needed, and then delegate the details of that to Sensing. As a result there is no place for a dedicated "Tasking for Sensing" specialisation, as all the sensing subject matter knowledge is within Sensing, not Tasks.

Another symptom of this bad practice is the passing of data through an intermediary component to give it to another; an example of this is illustrated in Figure 15. Components should access the data they need directly and not via an intermediate component. Higher level components are not a special case; the control architecture is about the level of abstraction of components, not the control of access to low level data.

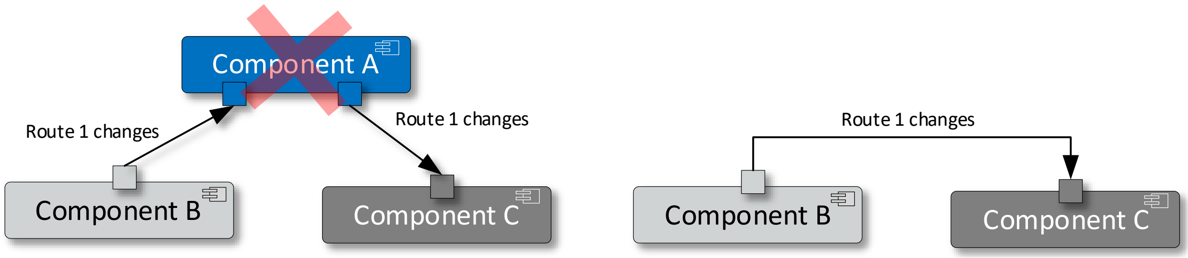


Figure 15: Information Is Passed Directly, Not Through Another Component

2.7.6 Do not use Tasks as a stand-in for planning

A common pitfall when determining the logical separation of the system is to look at a snapshot of the lifecycle of a component; a symptom of this mistake is to think that another component will provide the planning (e.g. Tasks). Tasks will provide the requirements for an activity, but this is about a high level abstraction, not about prior knowledge in time, as illustrated in Figure 16. If you find yourself duplicating data from a component in Tasks or a tactics extension then this suggests that you have failed to properly abstract up the activity to constraints and triggers.

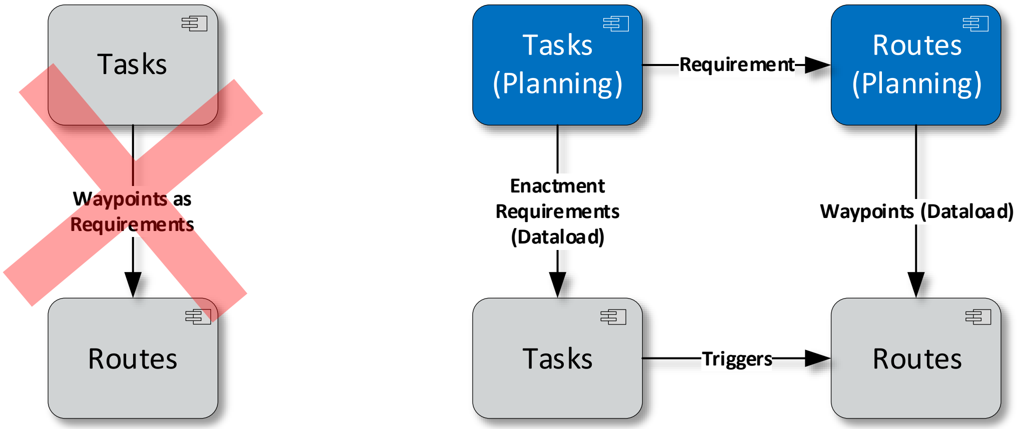


Figure 16: Tasks Is Not a Stand-In for Planning

2.7.7 Do not treat Tasks component as an overseer

A common problem is to think of the control architecture as a control hierarchy with Tasks sitting on the top as a substitute for a pilot in an overseer role. A symptom of this is long chains of status messages being passed between components, triggered from the same event, with Tasks acting as a progress tracker. The Control Architecture is about abstracted views of tasks; each component should only report progress in its own context. Any problems and decisions should be handled by the component that has enough information detail to correct the problem, which due to the subject-matter-centric nature of components will usually be in the one that detected the problem or change, if applicable to its role and responsibilities. Tasks will only be involved in the abstracted view of the actions, as illustrated in Figure 17.

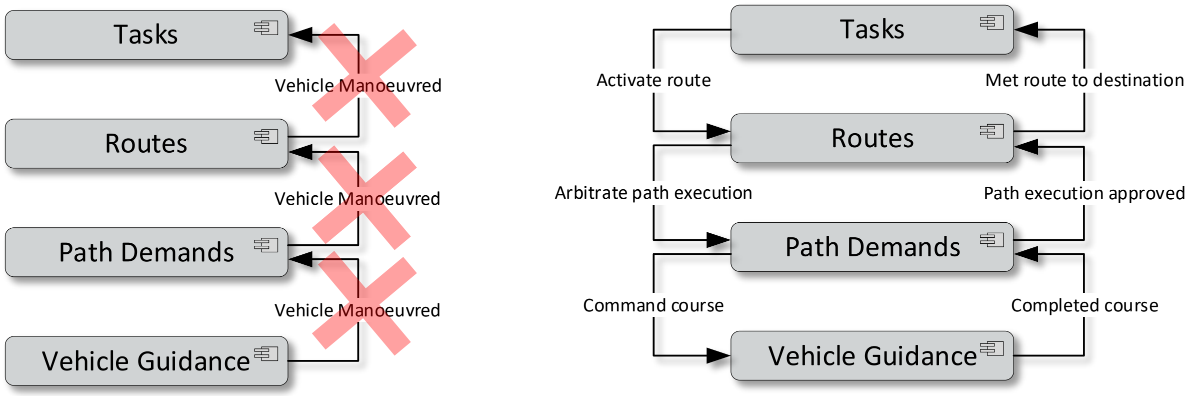


Figure 17: Reporting Between Components

Tasks should not need information about the progress of a component with the relevant lower levels of detail, but if it does, it can go direct to that component for the information. It does not need to be passed up a chain. Similarly, if a user wishes to view low level information about an activity this would be provided to the HMI directly from the related resource component.

2.7.8 Do not over define inter-component behaviour during system design

When analysing inter-component interactions in multi-component sequences for a use case, going into too much detail can lead to the component design being constrained to do things one way, which makes it harder to incorporate a change into the system at a later date. A symptom of this is showing detailed protocol handshaking in high level inter-component use cases.

Likewise, although the logic that is required from components should initially be determined assuming that all goes as desired, too much focus on multi-domain ‘sunny day’ and ‘rainy day’ scenarios may drive a component’s design to be overly specific and less resilient to change or reuse. Simple design patterns, for example not assuming that a request on another component is always carried out, can drive out a more general approach giving a generalised resilience.

2.8 Summary

System level artefacts that should be identified by this stage are:

- **Build sets** that define which components are required to meet the customer requirements; see section 2.2.1.
- **Component specifications**, which are populated with the parameters to be exchanged; see section 2.3.
- **Bridges**, which are populated with mappings that connect the services of interacting components; see section 2.4.

3 Platform Independent Component Design

This section covers the platform independent component design process, which considers how the PRA supports the maturation of component behaviour: the behaviour required from a component in order to fulfil its responsibilities within the system and provide its services. The maturation of this behaviour enables implementation of the component specification. Where compliant component specifications are available that meet the needs of the deployment, reuse of already implemented components should be considered.

The platform independent system design produces the component specification, which defines the purpose of the component, in terms of what it provides and what it needs from the system. This was discussed in section 2.3.

Component design is considered separately from component specification, as the specification should not expose any internal details and could be applicable to many different component designs. However, where the Component Designer discovers the need for new services, they should consult with the System Designer to agree any changes to the component specification. This will improve the correctness of the specification, and makes sure that component design remains within the bounds of the component scope.

The functional requirements placed on the components by the system design need to be analysed to further the maturation of component design artefacts. As with the system level, the requirements need to be analysed through a normal system engineering process; see Ref. [8]. In line with the rest of this document, this section assumes a separation of concerns from non-functional requirements, addressed in sections 4 and 5.

3.1 Component Modelling

The modelling performed by the System Designer will have defined the component specification, and this will have identified the component’s services along with the data required to support them (including data-types for parameters referenced by the service interfaces). Future systems are required to be rapidly adaptable at an increased frequency with an overall reduction in through life costs; this section covers what component modelling techniques support this goal, and what assistance is provided in the PRA.

3.1.1 Structural Analysis from Data

The PIM focuses on modelling high-level concepts to achieve the highest level of independence, as the level of independence inversely correlates to the amount of low level detail. Object-oriented design techniques should be used to define the system, for which additional guidance is given in Ref. [31] and Ref. [32].

A system wide data model can be used to help identify what data is available at the component interface. However, this does not provide a detailed structure of what the component needs to hold internally, which needs to be considered further to identify software structure. Typically this information is captured as a class diagram, and should address the following:

- Consider the “items” (the actualisation of the entities) that the component reasons about and manages; the semantics diagram within a PRA component definition can be used as a starting point for this by providing context to the component’s subject matter entities. The component specification will have defined the public entities, i.e. those entities needed to support the services. Those entities, within the subject matter part of the component specifications, can be considered as a conceptual definition for the “items”.
- Consider whether any additional internal entities are needed to support the public entities. The Component Designer should address this to enable the component model to be completed, and should also take into account the component’s scope boundary (as defined in Appendix B of the PRA Description Document, Ref. [2]).
- Consider the information and state data that entities need to capture. This is done by defining attributes on the classes to capture the data they own, or data they use to capture their current state.
- Consider the relationships between the entities. This defines which entities make use of or reference other entities. Note that references between entities are implied attributes – so there is no need to model attributes to show containment or aggregation of entities.
- Consider the operations needed to support the required behaviour. This includes operations needed to support services and any autonomous/internal behaviour. The level of behaviour detail needing to be defined at this stage will depend on whether code auto-generation will be performed at the Platform Specific Model (PSM) design stage, and on the selected middleware.

The structural and behavioural design of the component following standard object-oriented design practices may drive out additions to the data model to support data driving; examples include configuration data such as maximum and minimum limits of allowable values for a particular class. Methods for loading this configuration data should also be considered when modelling the internal structure.

3.1.2 Data Driving

The PRA promotes a data-driven approach in the development of the PIM; see the Data Driving policy. This means that data can be used to configure, initialise and determine how components can be expanded where applicable. The designer needs to consider whether a component would benefit from being data-driven through use of data in any of those ways, which are elaborated below:

- As configuration data, which includes datatype specifications, constants and constraints. This data defines the characteristics of the component, and the characteristics of types within its data model. Typically this data is used to instantiate specification-classes. These are classes which define the characteristics of a super-type; namely its class level attributes and value constraints which apply to all classes associated with the specification-class.
- As initialisation data, which provides initial state data used to instantiate entities according to their datatype specifications. The process of using initialisation data includes the loading of the initial data-set and population of any data dictionaries or enumerated types. Further detail on data loading can be found in the Data Load Interactions section of the Operational Support policy.
- As extension data, which determines how and in what way component behaviour can be specialised (e.g. identifying extension components and relating them to the entities they specialise).

Often a structure can be developed that allows for the data loading of an external data model, allowing for a design to quickly change as the data model develops. There are several standards specifically set up to support this; for an example see Ref. [26].

When looking at logical grouping of data driving, the *owner* of the data needs to be considered. Some data (e.g. software configurations) may be added by the component development team and baked in as part of the code level certification, other data will be set by the System Integrators, while still more will come out of mission planning. The consideration of all these data control points often leads to unearthing a whole set of data-driven parameters that may not otherwise have been identified. Although the data groupings are a PIM problem, how the different data controls are separated, including the impact on certification & accreditation, is a topic for the platform specific design (see section 5).

Note that using data alone to determine how a component can be expanded is covered by configuration data (through datatype specialisation) and initialisation data (by instantiating instances of specialisations). This can result in expansion of the range of entities that a component can manage and control, but cannot truly specialise behaviour beyond activating specialisations already factored into the component's design.

3.1.3 Behaviour Modelling

Static structural diagrams (e.g. class diagrams and entity relationship diagrams) only tell part of the story; it is also important to capture the behaviour expected from services. This is done using activity and sequence diagrams showing the relationship between external services and internal operations.

State transition diagrams should also be considered for classes that manage state data, showing how they will behave for different situations and operations; this can also help drive out additional required information. The level of detail to be modelled depends on how formal this needs to be, for safety critical systems this may well include action language statements providing a clear definition of the expected behaviour.

Use case diagrams are also important, as they help to represent the functionality of the component under analysis in terms of how it is intended to be used by external actors. Use case analysis is recommended as it is a modelling activity that is customer focused but component centred, and can capture the intended behaviour of individual components.

With this information in place it is then possible to design the test cases that will be used in testing of the component prior to integration.

3.2 Design Areas of Concern

Each component definition within the PRA contains design considerations and design assumptions sections. These outline key aspects of the behaviour, structure and dependencies that were anticipated when the component was being developed. As well as the component specific guidance, the PRA Description Document contains policies that are relevant to component development, and these are discussed in the following sections.

3.2.1 Data Configuration

When configuring a component using data-driven techniques, the specification of the component identifies the type of data that will be data-driven but does not concern itself with the internal structure that is required to achieve this. This ensures that the internal structure of the component is not restricted to any one design style.

Configuration is beneficial because it supports different sorts of deployments (e.g. different Exploiting Platforms or different hardware setups) and adaption to end-user scenarios (e.g. different role-fit, operational requirements, etc.). However, adding configurability will likely increase the complexity of verification, and associated cost implications should be considered when deciding whether to make a component configurable. For further guidance see the Data Driving policy.

3.2.2 Automation, Autonomy and Constraints

In order to aid understanding of the likely behaviour of a component, the PRA components have been grouped by levels of expected decision making responsibility and control. For further guidance see the Control Architecture policy.

Whatever the level of the component within the architecture, if it has the information needed to make an informed decision then it should make the decision rather than delegate it or push it up the chain to another component. This information may include the authorisation to make the decision. The level of autonomy that is supported, including requirements for when human interaction is needed, should be captured as part of a service contract. For further guidance see the Autonomy and HMI policies.

To support its decision-making, a component needs to provide the facility for the correct determination of constraints. Constraints may be provided as part of configuration data (see section 3.2.1) and loaded during initialisation, through planning activities, live via other mechanisms, or as an integral part of a component's data model. Component Developers should provide sufficient information to aid initial component integration, such as example constraint data. This will allow the System Integrator to better understand what is expected from the supporting components and infrastructure. For further guidance see the Constraint Management policy.

3.2.3 Capability and Health

It is important that where appropriate components are able to assess their capability over time and with use, including the ability to predict capability progression, and to highlight missing information that could improve the certainty or specificity of the assessment of their capability. This capability assessment functionality can support activities for alerting a user to anomalies, offering remedial action or performing automatic corrections.

As well as Anomaly Detection and Health Assessment components monitoring the health of the underlying infrastructure that a component relies on and is aware of, the component needs to be able to provide information for determining whether there are anomalies that are the result of systemic changes, which may be the result of faulty or corrupted software.

A Component Developer should provide data to support the configuration of health and cyber defence monitoring components (including detection of anomalies), allowing other components to better characterise changes in the capabilities of the system. For further guidance see the Capability Assessment, Health Management, and Cyber Defence policies.

3.2.4 Coordination between Nodes

The PRA components can be designed to cover the numerous types of situation or setting that the deployment may be involved in (e.g. training, maintenance support, and from mission planning to post mission data analysis). At the platform independent stage of design it is assumed that every component has access to the data that it needs to carry out its role; however, there needs to be consideration of how to coordinate data between multiple locations. Data coordination requirements are likely to drive out specific services and dependencies to be captured in the service contracts and service dependencies. For further guidance on coordination between vehicles in operation see the Multi-Vehicle Coordination policy, although note that this is only one specific case out of many types of coordination between nodes.

3.2.5 Safety Related Dependencies and Rely-Guarantees

In section 2.3.3 we covered definition of dependencies between services within the service specification. Now that the logical separation within the component is better understood, more detail can be added to this relationship. Rely-guarantees can be used to express the relationships between the services that a component guarantees and the dependencies it relies upon to provide those guarantees. A rely-guarantee method allows the behaviour of services to be abstractly specified, meaning that they can be verified separately without requiring extensive implementation details.

Where dependencies are expected to affect safety aspects of the system, rely-guarantees can be used to support modular certification. Safety information can be overlaid onto rely-guarantees to provide a safety view concerning the relationships in question, and they can be expressed at the desired level of component abstraction to support certification at that specific level. They can be refined for decomposed lower levels of design, down to the effect of individual code segments.

Goal Structuring Notation (GSN) may be utilised in conjunction with rely-guarantees to support safety certification (see Ref. [33]). GSN can be used to provide a reasoned argument that component guarantees will be met and provide the supporting evidence.

3.3 Identification of Extensions

In order to maximise reuse and reduce integration and testing overheads, the PRA allows for extension components, which can be separately modified, added or removed when needed. For an explanation of extension components and how this relates to the PRA, see the Component Extensions policy.

The designer should consider whether the component's behaviour benefits from being factored out into extensions. The System Integrator should be consulted about the areas of functionality that are likely to change through the system's life. This will help identify where use of extensions could reduce time and effort compared to that required when changing the whole component, due to the additional functional separation that they enable.

3.4 Good Practice

This section provides good practice recommendations applicable to typical cases of platform independent component design, including guidance for avoiding common pitfalls.

3.4.1 Do avoid pollution

Pollution is where a component has been expanded to the point that its subject matter now overlaps with the subject matter of another component. Pollution will result in a PYRAMID non-compliant component as described in Ref. [4]. The following points explain how to avoid common occurrences of pollution:

- Do not force details into a component that understands a subject more broadly so that it can "manage" the components that understand the specific detail. Components that understand a subject more broadly operate at a higher level of abstraction and should not know details.
- Do not duplicate information in one domain that belongs to another domain.
    - Use data in the domains without duplicating it; how this data will be accessed is a concern for the platform specific modelling.
    - Renaming the data classes is hiding the fact that you are copying data, and will lead to non-compliant components.
    - The components have been designed to be loosely coupled. If you need to duplicate data, checks should be made that you are not also duplicating functionality.
- Do not be too implementation specific in components higher in the control architecture; this may make them simpler, but will reduce the usability. They should stay at their designed level of abstraction to avoid pollution and promote reusability.

3.4.2 Do follow S.O.L.I.D guidance

S.O.L.I.D is an acronym for five of the key object-oriented design principles first put forward by Robert C. Martin, which should be considered when designing components; see Ref. [34] for this and further guidance:

- **Single-responsibility principle** – A service should have one job, giving it one reason for change. This principle also extends to classes within a component. This segregation will help the design in many areas, from partitioning the software to safety case creation.
- **Open-closed principle** – Components (and all other parts of the design) should be open for extension, but closed to modification. A key aim for a PRA deployment is to be open to change while still conforming to safety requirements applicable to its operational environment. This single principle can contribute significantly towards meeting this aim.
- **Liskov substitution principle** – Extension components should add to (or extend) a Parent Component’s behaviour, not replace it. This principle should also be considered in relation to polymorphism within the component.
- **Interface segregation principle** – A component should never be forced to implement a service interface that it doesn't use, and consumer components shouldn't be forced to depend on service interfaces they do not use.
- **Dependency inversion principle** – Components should depend on abstractions, not on concretions. In addition, abstractions should not depend on details, but details can depend on abstractions. This is about decoupling in component design, and should come from applying the open-closed and Liskov substitution principles. This principle should also be applied within a component.

Other reference texts with guidance for object-oriented design practice are widely available. For example, Ref. [35] provides general guidance on patterns and anti-patterns in object-oriented modelling of systems, whereas Ref. [32] provides specific guidance on object-oriented design in avionic systems.

3.4.3 Do design highly cohesive and loosely coupled services

The PRA components have been designed to each have a well-defined, tightly related subject matter. Designing highly cohesive components this way leads to the relationship between components being loosely coupled. This principle should be extended to the services. Services should be designed from a consumer focus, but also from the component’s perspective, i.e. consider a generic consumer for the component, but also consider the needs of the specific component under design.

Service design should be approached with care by those who are new to Service-Oriented Architecture (SOA), with specific attention to considering services from the consumer focus and understanding the difference between services provided by the component and those which are consumed by the component. Those new to SOA may be inclined to consider services as exchange of data, rather than requests for work to be done, which can result in poor service design.

3.4.4 Do design reusable services

The future of technology is hard to predict. Even if a service is only accessed by a single component in a build set of a specific design, it should still be designed so that it can be reused in the future, when previously unknown consumers are added to the system.

However when addressing services that you expect to be reused in various ways, care needs to be taken not to define a parameter or attribute that is so vague that it is meaningless. Care should be taken with quality and capability measures as they are particularly prone to this.

3.4.5 Do design services that are decoupled from internal logic

While the best component services are often developed in collaboration with Component Developers, the service interface should be abstracted from how the service is fulfilled. This allows for changes to the internal structure of the component to take place without impacting the interface, and vice-versa.

3.4.6 Do design composable services

Services can be composed of other services, which allows capabilities to be orchestrated at different levels of granularity. Composition in services can be promoted through common service patterns. General guidance on service patterns can be found in the Component Connections policy.

3.4.7 Do design independent services

To facilitate simple choreography services should be designed with the ability to be independent, having self-control over the logic they govern within an explicit boundary. Therefore, services should be designed to be able to stand alone from other services with the minimum of conversation between the consumer and provider (see Ref. [8]). This allows for a greater decoupling between components, and as a result greater independence in the development of components. Many one-way flow services with a high degree of dependency between them are a symptom that this has not been considered.

3.4.8 Do design state independent services

Services should minimise the amount of state information managed on behalf of consumers (users of the service). When a single transaction for a given consumer involves multiple service calls, there is sometimes a need for a service to manage the state for its consumer for the duration of the transaction. This need is greater in loosely coupled components, as state is private to the component providing the service. This maintains high cohesion and keeps consumer interactions simple, but is inefficient, scales poorly and is less resilient. To avoid these problems, services expect the transaction state to be maintained by the consumer. The consumer passes the transaction state into each service call related to the same transaction.

In the PRA modelling approach, this transaction state can be maintained in a counterpart class that holds a counterpart relationship with the corresponding counterpart existing from the provider component's perspective. The consumer component's counterpart class is a representation of its understanding of the transaction. This counterpart will then be passed in any service calls and translated by the bridge into terms understood by the component providing the service. Following this approach will reduce resource consumption and enable the service to efficiently handle multiple concurrent consumer requests.

Symptoms of this principle not being followed are the need to capture states and mode information in bridges (making them over complex and potentially polluted by component subject matter), or having to pollute another component with knowledge of the component in question's state. Over-complex bridges that have a high Development Assurance Level (DAL) potentially make certification more difficult, among other issues.

Often, to achieve state independent and atomic services it is necessary to pass more information in the service operations in order to avoid reliance on state management between calls. This leads to more complex service operations (see the Component Extensions policy, which describes extensible components for which extension data is usually data-driven).

3.4.9 Do consider mission planning

Part of the separation of concerns between the platform independent and platform specific models is that the Platform Independent Model (PIM) assumes that there are no interface delays, all exchanges are instantaneous, and all data that is in the domain of the component exists in the component. That is not to say that interface latency, data transfer synchronicity and data loading are not all important, but that they are considered independently during the design. This seems to cause some particular problems when considering mission planning, which in the PIM is co-located with the operation; planning and execution are not fundamentally different. The following guidance is recommended:

- Do not treat mission planning as solely an off-board process, all execution involves some level of planning.
- The parts of a plan that are either requirements or solution options should be clearly and distinctly defined as such. This enables design that can allow plans to be changed appropriately as circumstances change.
- Do not assume that smart equipment is unable to support extensive re-planning on the fly.

Note that maintaining the level of abstraction of components will promote reusability, as components will support both planning and in mission use.

3.5 Summary

PIM artefacts that should be identified by this stage are:

- **Classes** for each component, depicted on class diagrams, which represent the attributes and operations required to support the services of each component, the operations associated with those services, and the relationships between the classes.
- A **data driving** model, representing any data driving requirements.
- **States** for classes whose behaviour is state-dependent (usually depicted using state diagrams) which represent the sequences and cycles of states entered by instances of the associated class, as the class progresses through its lifecycle. Note that state here refers to states which control behaviour or lifecycle, and not simple changes to data values.
- **Extensions** for some components, including the service interface with the parent.

4 Platform Specific System Development

The principle behind the PYRAMID approach is a separation of concerns by only dealing with the problems that are specific to the model abstraction level being considered at the time. The platform independent modelling looked at the functionality that was required without consideration for the execution platform specific details of how it would be achieved. The next step is to consider the non-functional requirements and the infrastructure architecture strategies, while still remaining independent of the detailed mechanisms of the underlying middleware that is to be used to achieve the desired result; this will constrain the system more and add detail.

The infrastructure needs to be identified so that it can be related to the platform independent system design outlined in section 2. This section will focus on the platform design decisions and the resulting impact on the components.

Although it is possible to miss out the modelling of the Platform Specific Model (PSM) and go straight to the application software code, the underlying principle of *separation of concerns* allows for solutions to problems to be considered in a narrow context, avoiding overcomplicating parts of the design stages and missing the impact of some of the decisions.

4.1 Non-Functional Requirements

The specific Exploiting Platform infrastructure choice will mainly be constrained and selected based on the non-functional requirements of the deployment system (e.g. environmental and maintainability requirements). It is not unusual for the Exploiting Platform infrastructure choice to be heavily influenced by the customer community, either through a preferred supplier or existing standards. Figure 18, adapted from Ref. [36], shows sources of non-functional requirements.

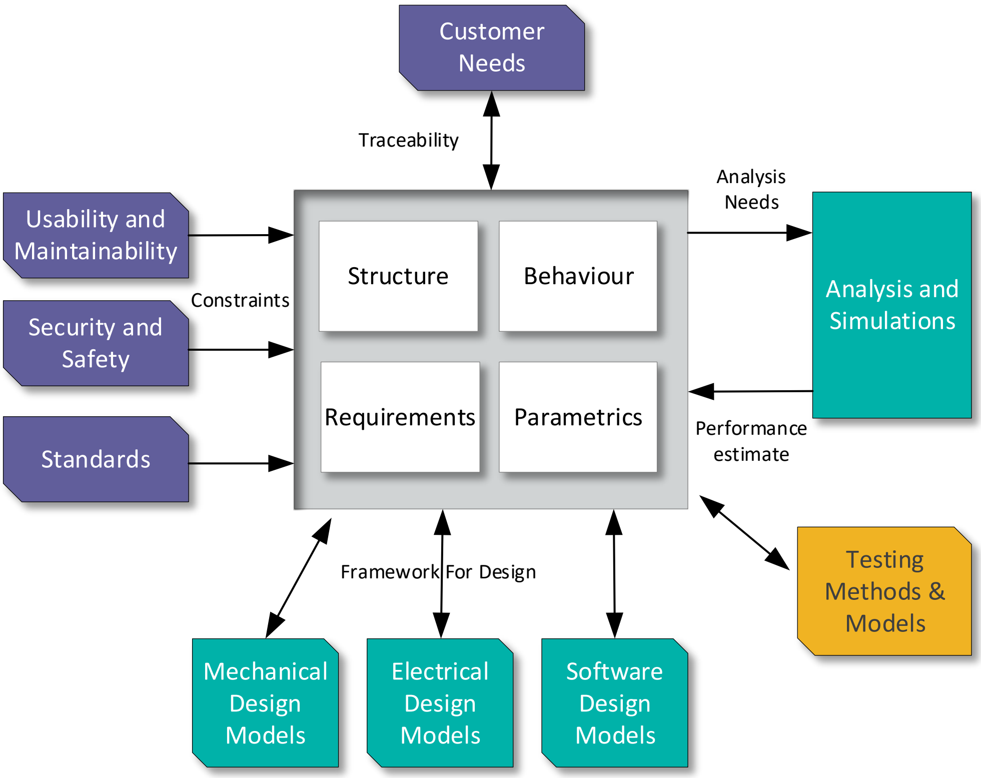


Figure 18: Different Sources of Non-Functional Requirements

The non-functional requirements will need to be analysed and considered in conjunction with the considerations outlined in the subsequent sections of this document.

4.2 Infrastructure Architecture

An Exploiting Programme needs to make a number of decisions around the specific infrastructure that supports the components it uses. Some of these decisions will be based on the scale of the deployed system, and whether the system needs to be distributed across a number of computing resources.

The complexity of the architecture transformation, and so the amount of effort required, depends on the complexity of the target of the transformation; for example, if the infrastructure needs to support deployment of the components across a distributed computing architecture then that is more complex than deploying to a single piece of executable software running on a single computer. Additionally, the infrastructure that supports the components is typically a software infrastructure including middleware, operating system and hardware (see section 4.2.1), but depending on the implementation it may include Field-Programmable Gate Arrays (FPGAs), crystal oscillators, integrated circuits or relays.

This section outlines some specific infrastructure-related areas that need to be considered at this stage of the design, and points to guidance within the PRA Description Document, Ref. [2].

4.2.1 Layered System Infrastructure

A layered infrastructure architecture is recommended for a deployment where possible, as this allows the application software to be decoupled from the underlying infrastructure. This layered approach employs a well-defined interface between the applications and the infrastructure. The number of layers in the infrastructure is dependent on the precise implementation; Figure 19 illustrates an example layered approach.

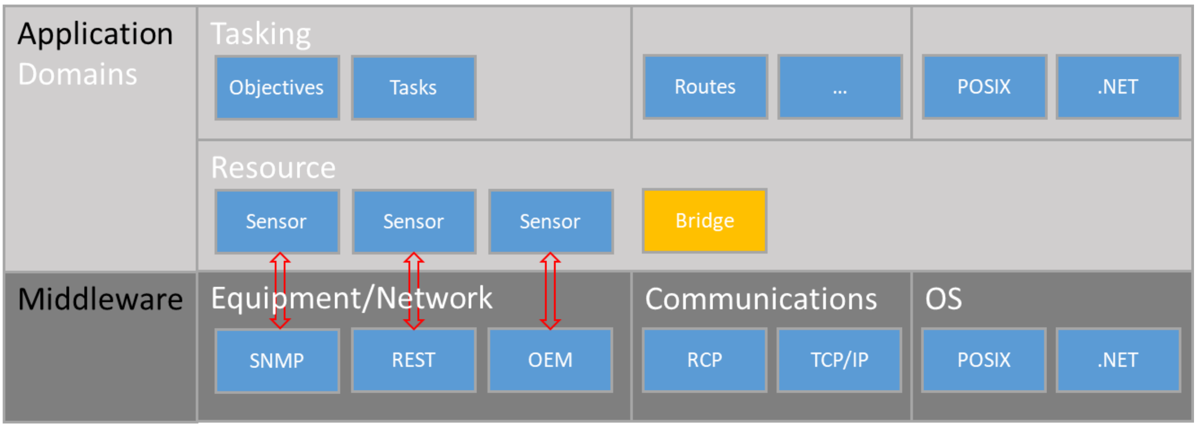


Figure 19: Example Deployment Architecture

Communication between components should use bridges to connect their defined service interfaces. However, a component is allowed to access middleware directly for specific purposes within its subject matter, such as:

- Reading the system clock;
- Controlling or interrogating a hardware resource;
- Inspecting health data.

Access to middleware must never be:

- Beyond the scope of the component's subject matter;
- Instead of using the component's defined interface.

It is expected that an Exploiting Programme will use an industry standard Application Programming Interface (API) for access to middleware. For further guidance on a layered architecture see Ref. [26].

4.2.2 Support for Scheduling

An infrastructure architecture for a deployment must consider a run-time system that is capable of scheduling. This includes not only scheduling of the software items but also threads within those (i.e. supports software items being multi-threaded). Different types, extensions and combinations of scheduling algorithms should be considered (e.g. earliest deadline first, shortest job first) when choosing the work scheduling method. Scheduling of access to resources is detailed in more depth in the Resource Management policy.

The software infrastructure must also provide a mechanism for implementing threads and semaphores for protection of multi-threaded code. For more information on model based design for real-time systems and the importance of thread management see chapter 8 of Ref. [8].

4.2.3 Software Resource Allocation and Protection

The expectation is that some parts of the deployment system will be safety critical, and so the software infrastructure will need to support safety critical software. In each case, the infrastructure architecture must consider the safety level of the software it supports.

A PRA deployment needs to consider software resource allocation, and protection between software items or between groups of applications (protection domains). Software items should not be capable of accessing the memory space of other software items.

All communications between applications (except those in the same protection domain) should go via the software infrastructure. For lower safety criticality levels, dynamic memory allocation should be possible, and this should be constrained to avoid interference between applications.

4.2.4 Time

An Exploiting Programme may have a requirement for a common representation of time across system nodes (e.g. multiple air vehicles or items of hardware within an air vehicle). When choosing a time synchronisation method, the system design should take into account its use of the infrastructure as a time source.

4.2.5 File Access

A PRA deployment must consider a file access mechanism for software items, which should allow both read and write access to files, and should be agnostic of the hardware mechanism for data storage. These requirements apply whether data storage is provided on each line replaceable unit or centralised within the system. For additional guidance see:

- Recording and Logging policy.
- Storage policy.
- Cryptographic Management interaction view – for protection of files accessed by different applications.

Note that protection of file access should be thought of both in terms of read (access should only be given to the appropriate applications) and write (a component should not be able to use more storage than is allocated to it).

4.2.6 Human-Machine Interface

A PRA deployment should, as required, provide a mechanism for generating information such as visuals and audio for Human-Machine Interface (HMI) output devices (e.g. rendering a graphical user interface to a display), and for obtaining user interaction events (e.g. pointer movement) from HMI input devices (e.g. keyboards, mice, hands on throttle and stick (HOTAS), touchscreens, etc.).

It is expected that as a minimum any graphics support should be through a primitive interface; higher level functionality can be achieved using graphics libraries and toolkits. For more information see the HMI policy and interaction views, Ref. [2].

In a similar way, a basic mechanism for capturing and playing audio may be needed; this audio can range from basic sounds (e.g. warning beeps) to speech and complex audio sounds. For more information see the HMI policy and the Human Communications interaction view, Ref. [2].

4.2.7 Health and Usage Monitoring

System infrastructure faults (e.g. power failure) potentially impact the execution of every application on the Exploiting Platform; see the Cyber Defence and Health Management policies. Some of these faults (and resulting error messages) may occur because the system is being initialised, while others may not get reported if the system is being initialised or is incorrectly configured. These occurrences are often an indication of bad system design.

The Exploiting Platform infrastructure needs to provide a mechanism for reporting faults found in the processing hardware, and a mechanism for reporting faults found in the execution of the software also needs to be provided (e.g. invalid memory access, arithmetic errors etc.); for more information see the Middleware Error IV (Appendix C of Ref. [2]) and the Health Management policy.

4.2.8 Software Configuration

Where software configurations refer to data that is related to the subject matter of the relevant component, this should be treated as data driving within that relevant component. The Data Driving policy should be referred to in this case. Where software configurations refer to software behaviour, e.g. buffer sizes, this information should be included at the infrastructure level, but may also have to be included in subsequent processes for achieving certification or accreditation relating to the software.

4.3 Certification and Accreditation

Whilst safety and security assurance objectives are different, the organisational and procedural issues are very similar. Safety and security assurance are properties that only exist for an entire product. An Exploiting Programme will have the responsibility for the analysis and verification activities, and completing the certification and accreditation. For this reason, exploiters should follow their normal safety and security processes; this section provides guidance in support of this and highlights where the PRA contains safety and security related information for components.

The platform specific design needs to provide deployment-specific safety and security isolation. The deployment process needs to be flexible and support a number of strategies to allow these requirements for isolation of information to be satisfied. This flexibility can be enabled by segregating components so that certain data and functionality are partitioned, and then producing the data that enables the infrastructure to instantiate the required segregation. Segregation may be done for any reason, and by any means.

An Exploiting Programme should segregate components based on the isolation requirements identified as part of the platform specific design; this should be in line with the following recommendations:

- If segregation is used, a component instance should be kept wholly within its partition.
- An extension is treated as a component in its own right, subject to the rules defined in the Component Extensions policy. The PRA does not preclude an extension being segregated from its parent or other extensions to the component.
- A component could be deployed in multiple instances that could each be segregated from one another.

If a platform specific design has no isolation requirements then the mapping can be performed on a purely resource allocation basis. A further complication that needs to be taken into account is that some other non-functional requirements might conflict with isolation requirements (e.g. performance related concerns).

While partitioning was not considered during development of the PIM, platform specific security and safety requirements may lead to changes in the build sets that take into consideration the new partitioning, as illustrated in Figure 20.

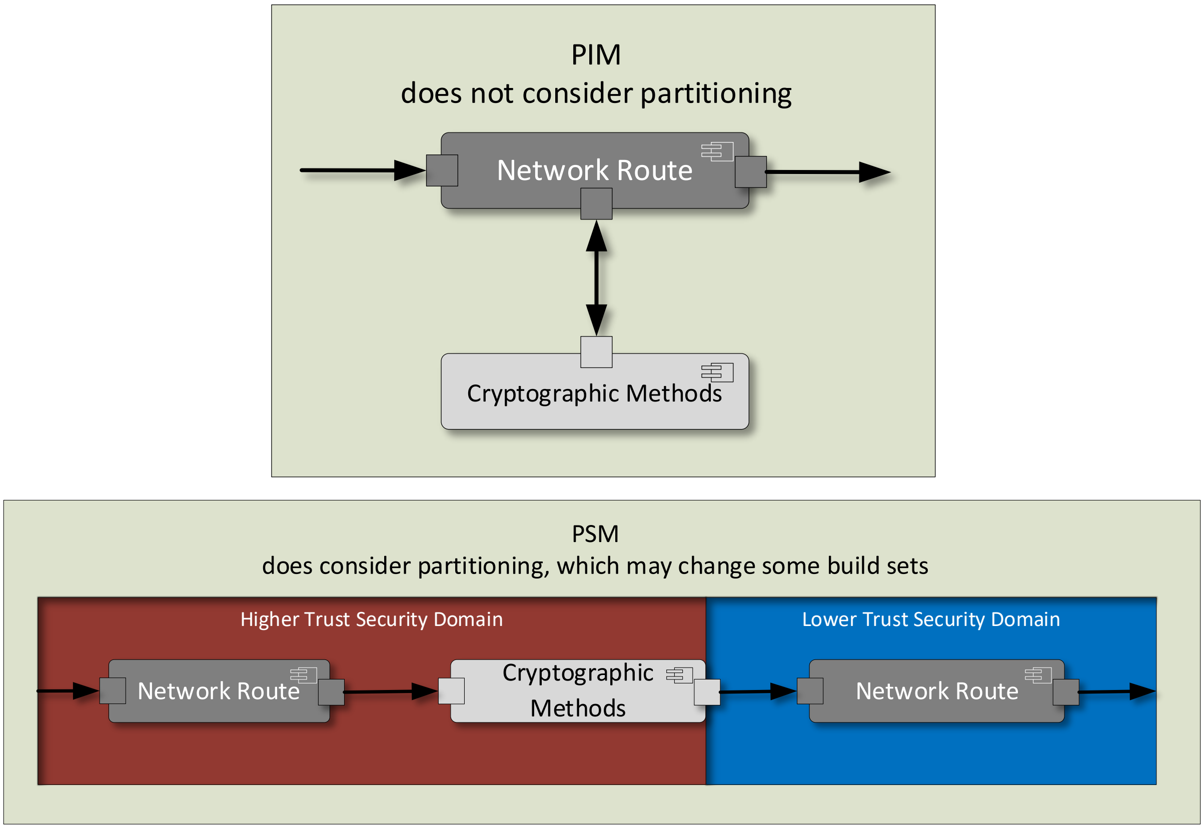

PIM
does not consider partitioning
Network Route
Cryptographic Methods
PSM
does consider partitioning, which may change some build sets
Higher Trust Security Domain
Lower Trust Security Domain
Network Route
Cryptographic Methods
Network Route

Figure 20: Example of Partitioning for Security Purposes in the Platform Specific Model (PSM)

Both security and safety requirements may drive certain components to be high assurance. DO-326A, Ref. [21], covers data requirements and compliance objectives, related to aircraft safety, for aircraft development and certification. Ref. [37] and Ref. [38] tackle the security assessment process and safety process.

4.3.1 Safety Considerations

The PRA has been designed with safety certification in mind; each PRA component has been assigned an indicative Item Development Assurance Level (IDAL), which is the highest IDAL that the component is expected to have to achieve for a large (>5700kg) air vehicle (see the Safety Analysis policy). The IDAL and associated analysis are provided for design assistance only, and do not place a requirement on the Exploiting Programme.

4.3.2 Security Assurance Considerations

Security controls should be identified as a result of a security risk assessment. Technical security controls are either security enforcing functions (SEF) or security related functions (SRF), and these are identified through a low level assessment and classification of data flows between components. A security considerations section is included within every component definition in the PRA, which helps identify whether the component is expected to participate in security related functionality. The security levels and associated functions are provided for design assistance only and do not place a requirement on the Exploiting Programme.

A Security Guidance for PYRAMID Exploiters document, Ref. [60], is available from Dstl on request. This guidance document provides additional insight on a number of security aspects that are likely to apply to Exploiting Programmes using PYRAMID-based deployments, including those "good practice" engineering and development standards necessary for a secure implementation, and other considerations to be addressed in order to achieve security accreditation and capability assurance sign-off for MOD acquisition projects.

4.4 Data Management Design

The PRA design is predicated on the concept of data-driven components (see section 3.2.1), which allows existing components to be reconfigured for reuse in new systems. This configuration exercise is in addition to the run time flexibility also provided by the data-driven paradigm. Following the PRA guidance should leave what are in effect placeholders for data driving in the deployment. The mechanisms for this data driving need designed, and data management design covers this.

4.4.1 Reconfiguration

A deployment needs to consider a mechanism for reconfiguring components, at runtime, to one of a predetermined set of configurations. This reconfiguration should be as transparent as possible to components which are not affected by the reconfiguration. For more guidance see the Data Driving policy.

Reconfiguration at runtime needs to be very carefully considered so that the reconfiguration does not result in safety or security issues arising as a component transitions from one configuration to another, or for the transition to be erroneously detected as an anomaly due to an error.

4.4.2 Data Synchronisation

A data synchronisation strategy enables multiple instances of a component to be deployed and segregated whilst acknowledging that an amount of data needs to be synchronised between instances.

Reasons data synchronisation is required include:

- Component instances are separated by mission phase (e.g. mission planning and mission execution) in different locations.
- Load sharing, e.g. to stop bottlenecks of resource usage at a particular component instance and/or to optimise collective performance.
- Multiple instances of a single component existing, e.g. for security partitioning, or physical segregation for safety redundancy (see section 4.3).
- Components that exist explicitly for the purposes of data replication, e.g. where parts of a system are aligned for backup or redundancy. Communications components could be used for this purpose; see the Use of Communications policy for more information on how these components can be used.

Partial replication may also be desirable when implementing components into different situations or settings from one another, i.e. a component can be coordinated between multiple air vehicles, each with their own version of the component sharing some data.

Services need to be defined to support data synchronisation; these should follow pre-defined patterns, e.g. ‘peer-to-peer’ synchronisation, or synchronisation via a common data source (e.g. database).

Data synchronisation between components presents an overhead; the choice of pattern needs to account for the ways in which the components could be deployed (distributed or not) in order to take account of system constraints (e.g. bandwidth and latency).

4.4.3 Inter-Application Communications

Inter-partitioning and inter-computing resource communications mechanisms need to be provided by the software infrastructure. Each component’s provided and consumed services are connected by bridges, which are able to access the underlying communications mechanism provided by the software infrastructure, allowing the components to be distributed across the available computing resources.

The configuration of the partitioning and communications mechanism in the bridges could be data-driven to enable flexibility in the process of deployment. Even in a single Exploiting Programme, a number of deployment configurations will be required to support different phases of system and integration testing. Implementing a data-driven configuration approach enables tools to be developed to aid the deployment process.

Note that when assessing the integrity required to achieve the desired safety or information assurance goals, the level of assurance required for the bridges between components will also need to be assessed.

4.4.4 Post Mission Data Analysis

Components should log significant decisions taken by the component and any events that the component raises or receives, which will allow the actions undertaken by components to be analysed during post mission analysis. Logged data also provides a record of how components contributed to outcomes, which can be used as evidence in cases where mission events have legal consequences. For more guidance see the Recording and Logging policy and the Operational Support policy.

4.5 Component Services

4.5.1 Service Deployment Design

Knowledge of the current and future Exploiting Platforms that may harness a deployment should be utilised to develop component services beyond their specification in the PIM. The mode of inter-component communications is a fundamental characteristic of the interactions, and the two modes that are typically chosen between at this stage are synchronous (request response) or asynchronous (event driven). The choice of communications mode is likely to be based on the environment that the system is to be used in; see Ref. [13].

A deployment's services, including its internal services, should be available to System Integrators outside of the original deployment so that it is forward compatible. A PRA-based system should be able to discover which known services of systems and equipment external to the deployment are available at a certain time, as covered in the Capability Assessment policy. For more information on component discoverability see Ref. [11].

Attributes on service interfaces are expected to be made specific to a deployment. An attribute on a PRA component may be specialised into one or more deployment-specific attributes. Service interfaces themselves are also expected to be made specific to a deployment, e.g. either the name or description made specific to the deployment while remaining within the PRA component's subject matter.

4.5.2 Architectural Alignment and Interoperability

Architectural Alignment is the process of aligning a deployment's architecture to the architecture of other (external) systems.

This alignment supports the sharing of best practice and the pursuit of interoperability through the use of patterns and consideration of component responsibilities and data exchanges. For an example of how to support interoperability with other systems, see the Tactical Exchange interaction view.

When considering other architectures, it is important to be cognisant of the roles and responsibilities of PRA components in order to ensure that alignment with external components doesn't conflict with PRA component definitions. Although there may be little or no control over external components, the internal alignment should not blur component boundaries or introduce tight coupling.

4.6 Infrastructure Implementation

As part of the PSM the technology dependent detail needs to be captured and included in the design; this includes identification of low level communications protocols and underlying operating systems and libraries. The design abstraction should not be confused with ambiguity; from a system point of view the aim of the modelling of the Platform Specific Implementation is to generate an unambiguous definition of what the component interfaces are, and what the component requires from the infrastructure. By working with Component Developers the service definitions can be aligned, reducing the bridging effort.

Safety and security targets still need to be met in the infrastructure that supports the components; when implementing a bridge, the DAL of the components whose service interactions the bridge supports need to be considered. Operating systems and middleware also need to support the DAL requirements of their hosted components.

4.6.1 System Reuse

There is always more than one route to a final product and a new system is never designed in a vacuum, therefore there will be existing systems that can be in part reused within a PRA deployment. Reuse can take the form of code reuse or system design reuse; either way the blocks of code need to be considered. PRA components are deliberately designed to encompass isolated areas of functionality to increase the potential for reuse, making it likely for legacy code to encompass the functionality provided by a set of components. Although it is the intention that artefacts can be reused at any stage of the design process, reuse of PSM artefacts is typically more difficult due to the greater detail of platform specific specialisation and dependencies by that stage. Therefore if it is an aim is to generate feasibly reusable PSM artefacts, both those working on the PIM and PSM should consider how their decisions may affect this reusability. Most decisions about the functionality of reused artefacts are appropriate earlier, at the PIM level, before these decisions are implemented at the PSM level.

It is important to note that reuse of developed components may adversely affect the security of Exploiting Platforms; the use of a developed component (or fragments thereof) on both secret and lesser classified platforms has the potential to compromise the secret one. Care must be taken in this respect, and the Security Guidance for PYRAMID Exploiters, Ref. [60], has further information on provenance and proliferation of developed components.

4.6.2 Model Transition

The steps outlined in section 4 can be viewed as a generic set of guidelines and recommendations applicable across all components that map the PIM to the PSM. As these guidelines are disconnected from the middleware solution and final messaging protocols, they can be captured in a transformation specification, which itself forms the requirements for a model translator. The model translator can then be used in a transformation engine to do the mapping and automatically generate the PSM, as summarised in Figure 21, which has been adapted from Ref. [39].

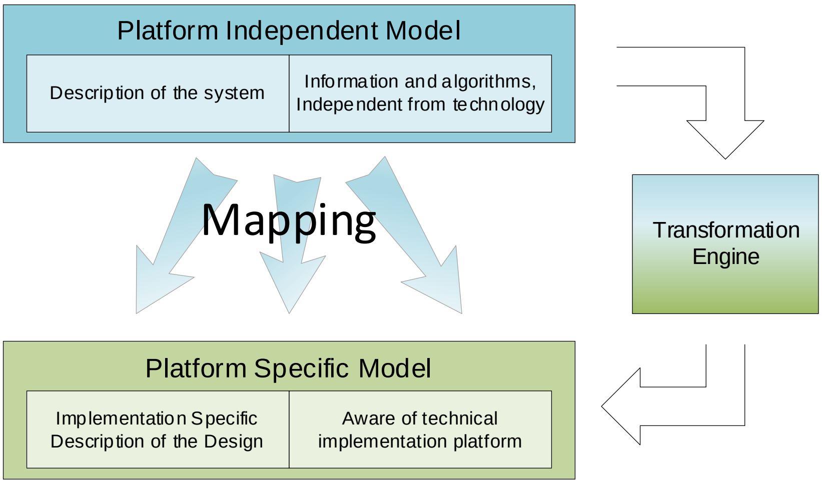

Platform Independent Model
Description of the system
Information and algorithms, Independent from technology
Mapping
Transformation Engine
Platform Specific Model
Implementation Specific Description of the Design
Aware of technical implementation platform

Figure 21: PIM to PSM Mapping

Automatic transformation of PIM to PSM ensures a consistent approach for all components, and allows the impact of changes to the PIM to be captured. A PSM can still be inspected, allowing any inspection of the detailed design before it is committed to code. This is particularly important in high assurance level implementations.

Different transformation specifications can be used to generate different PSMs for different target systems, allowing a family of PSMs to be generated for different uses. Automatic transformation can be taken a step further with the PSM being used to automatically generate code for the structure of component implementations and the integration of components with the Exploiting Platform infrastructure.

The adoption of standards based shared meta-models across the deployment is more likely to allow for the sharing of information between design tools supported by different vendors. If this approach is adopted across multiple deployment programmes then they can be quoted along with the PRA model to define a de facto standard. For example, by using the FACE standard (Ref. [26]) it is possible to:

- Map conceptual data elements to the deployment-specific elements, automatically identifying logical and physical representations on all sides of an interface linked by a common conceptual meaning.
- Select suitable logical representations or translations that reflect physical phenomena from a library of predefined logical data types. This can support future reuse.
- Select a subset of logical representations from the existing library, which will reduce the number of logical conversions needed, and will support the maintenance of bridges within the deployment.
- Generate interface control documentation that matches the modelled interface definition. This is particularly powerful where the final interface is also auto-generated from the model. If all documentation is automatically generated from the model, this should help avoid the staleness that often comes during later revisions of the product where the documentation is not updated.

4.7 Good Practice

This section provides good practice recommendations applicable to typical cases of platform specific system development, including guidance for avoiding common pitfalls.

4.7.1 Do bridge through middleware

What is treated at the platform independent level as a simple peer interaction may occur between physically segregated areas, as illustrated in Figure 22. Part of the job of bridges is to hide the complexity of communications from components. A component should be unaware of the source of information, although the service implementation needs to be designed to cope with latencies or delays resulting from sourcing information from a distance away.

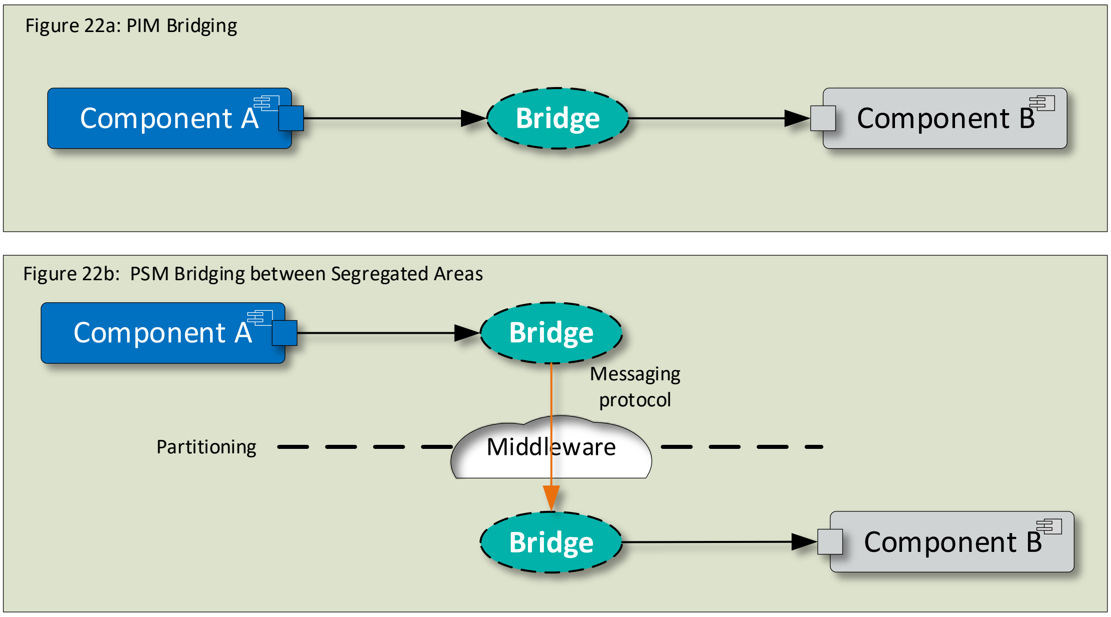


Figure 22a: PIM Bridging

Figure 22b: PSM Bridging between Segregated Areas

Figure 22: PIM Bridging (22a) and PSM Bridging between Segregated Areas (22b)

The concept of a service being separated from the internal working of the component is to allow the component to carry on activities while the related service deals with the bridge and any data exchange. If components cannot be decoupled in a particular platform, consideration should be given to co-locating them when deploying them.

Additionally, where PSM bridging is used between segregated areas particular consideration should be given to the effect on the component's operation if the interconnection is lost.

A common middleware interface / messaging protocol should be used to aid with cross platform re-usability; see section 2.7.1 for more on the use of additional standards and reference models.

4.7.2 Do not use component variants to cater for specific uses

The PRA components have been based on a logical definition of what the component needs to do, and what data it needs to do it. Where the same logical activity would be duplicated in a legacy functional decomposition (e.g. in different subsystems), there is only one component definition in the PRA. An example of this would be the Observability component determining if a communication signal is detectable for both survivability and for communication uses.

The goal is to break away from stove-piped designs, into something that is more flexible while still meeting high security and safety standards. It is possible to recreate a legacy functional decomposition by using component variants (different specific use implementations of the same component); see Figure 23. This should be avoided. Other choices such as using a shared component as a resource, using a component with tailored extensions, or multiple copies of the same component should be considered ahead of creating a variant of a component for a specific use.

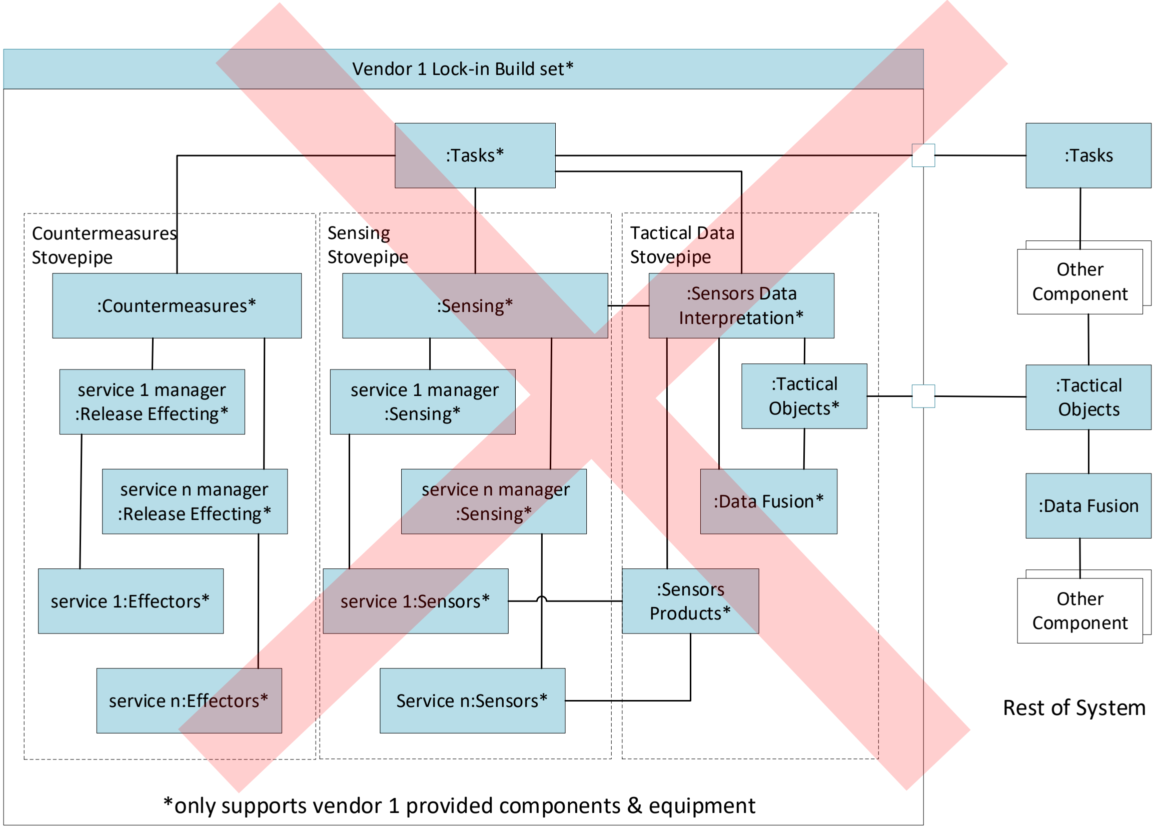


Figure 23: Do Not Use Component Variants as a Tool to Promote Vendor Lock-In

4.7.3 Do use component variants where functionality is separated between platforms

Following on from section 4.7.2, where component variants should instead be used is where the functionality is being separated in space or time. At later iterations of the design, after the required design separation is understood from the platform specific analysis, the functional model can be updated to reflect this. Typical examples of this would be offline planning and maintenance support, both of which are typically deployed on different hardware to an air platform, with different design goals. The partitioned functional model is technically platform specific and it is recommended to keep this separate from the PIM, so that unexpected logical behaviours detected in testing can still be analysed in either the PIM or a fully separated functional model, as appropriate.

Note that as the PRA does not include definitions of services to support splitting of a single component's functionality, additional services may need adding to components to enable interaction with other variants of the same component, for example to synchronise data.

This may include data loading services, such as where a mission planning variant's data needs loading into an operational variant. To aid reusability, it would be good practice to develop these types of service in-line with suitable standards.

4.7.4 Do separate the concept of ‘what data is’ from how it is acquired

The PIM assumes that there are no interface delays – all exchanges are instantaneous and all data that is in the domain of the component exists in the component. These things do however need to be considered as part of the platform specific modelling, including where data is created. It is the responsibility of all the components to manage their own data, including data loading; see the Operational Support and Storage policies. However, a common approach to data loading should be identified and implemented at the PSM level, as it will be infrastructure dependent.

4.7.5 Do use counterparts

The principle of components dealing only with data within their own subject matter knowledge means that different components are likely to have their own views on the same real world thing, such as some liquid fuel on an Exploiting Platform. The Fluids component may hold data on the volume of the fuel, whereas the Mass and Balance component is concerned with the mass of the fuel, and Environmental Conditioning may view the fuel as a heat source or sink. In such a case, each component contains a class to represent the fuel from the component’s subject matter specific point of view, with these classes being used to form a counterpart relationship. See the Component Connections policy and Ref. [9] for more details on counterparts.

Counterpart relationships support separation of information between the relevant domains, helping avoid subject matter pollution. Key benefits of such an approach are summarised below:

- The domain based approach to components is predicated around minimum amounts of information being passed between components because each component should encapsulate information relevant to its subject matter. However, it is the nature of an integrated system that there will be related changes in different components; counterparts are a good way of making sure that this related information is kept aligned.
- Information on an object known to the deployment can be obtained efficiently, with relatively low data passing between components; for example through the use of system wide identifiers to obtain the relevant counterpart data from the owning components. This also applies, in a broader sense, to management of counterparted information, potentially via standard operations.
- A counterparting approach can help link preparation data to associated execution data between components (see section 4.7.6).
- By mapping counterparts (via bridges), their data can be synchronised between domains to enable coordination and consistency across a deployment.
- Use of counterparts supports the extensibility and scalability of a deployment, as it allows easy addition or removal of information types made up of data linked through counterpart relationships. It enables specialisation of shared interfaces between domains, regardless of programming language and execution platform.
- Counterparting is a suitable way to enable data views (see section 2.6.1.1), since counterpart relationships can indicate the attributes that a component may need to make available to other components.

Note that while a counterparting approach is effective for capturing relationships between the information in different subject matter areas, a service operation approach can be advantageous when capturing event-based relationships, such as command and control relationships.

4.7.6 Do coordinate preparation and execution interactions

As part of the platform specific development, concerns around how to manage resources and latencies need to be taken into account when considering component interactions. There is a particular issue in how prepared resources are matched to their intended usage at execution.

The solution is likely to involve an action level identifier, for the planned action, that is a counterpart to an identifier at the resource level; see the Component Connections policy and Ref. [9] for more details on counterparts.

The counterpart identification cannot be determined implicitly in bridges as this is a problem of recall of earlier preparation. If the identifiers were contained within the bridge, the user of the resource would still have to identify something to the bridge that would enable it to recall the relevant planned action. This means the System Designers need to consider how identifiers are managed as part of component design and capture this. Otherwise, different parts of the system will not work together, as show in Figure 24.

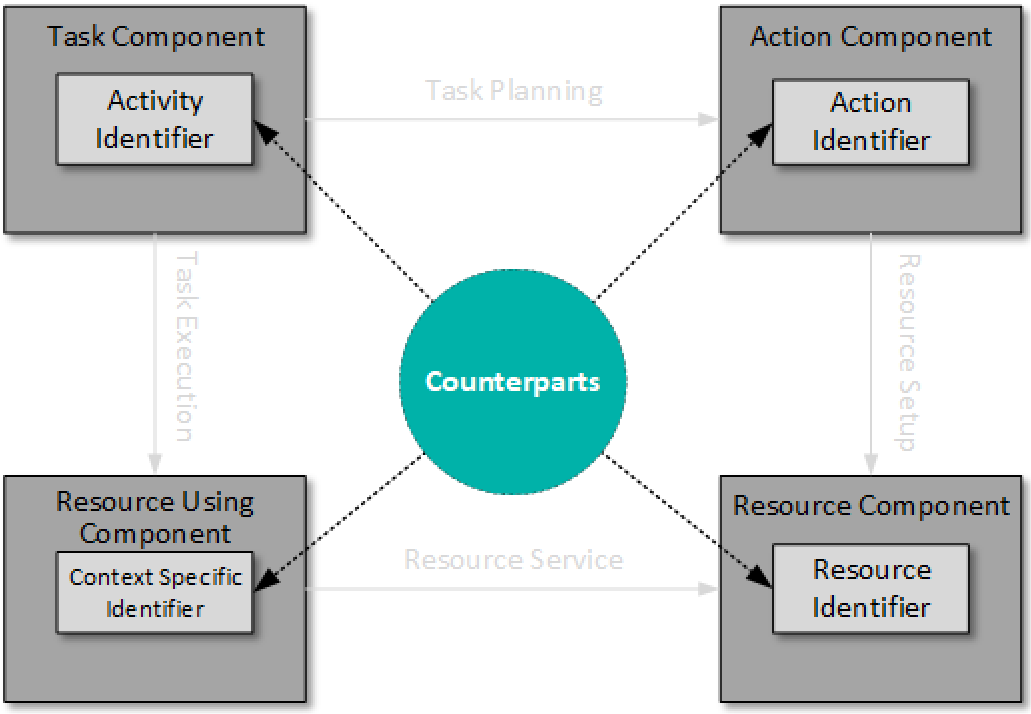


Figure 24: Counterpart Relationships

4.7.7 Do capture and share platform specific information

It is important that platform specific information is captured and shared early in the process. This enables component design and development to consider platform specific aspects when needed, and is especially important for aspects common across components. Artefacts such as standard documents or shared models may be used to capture this information into a single record source, which can then be referred to when needed in specifications.

4.8 Summary

An iterative process is expected to be followed that will involve looking at the execution environment and its impact on component design specialisation and separation (see section 2.3), as well as the integration of components as they are produced (see section 5). As the design gets closer to the implementation, the specific additional contribution of the technology agnostic architecture becomes less clear. However, the core goals, especially around reusability and extensibility, should be kept in mind.

Artefacts identified at this stage should be appropriately captured (e.g. in platform specific equivalent versions of a component specification) and shared with Component Designers and Component Developers. The mechanisms for this should enable close regular interfacing between any separate parties. Some of the typical artefacts identified at this stage are:

- **Non-functional requirements** to be met by the system; see section 4.1.
- **Infrastructure definition**, which specifies the sub-systems, infrastructure interfaces, middleware and operating system chosen to support the components; see section 4.2.
- **A partitioning approach**, which defines how components should be segregated or aggregated to meet safety, security and performance requirements; see section 4.3.
- **Configuration**, preparation of data that supports the data driving design decisions; see section 4.4.
- **A component services approach**, which defines the underlying protocols and mechanisms that have been chosen to tie services together; see section 4.5.

5 Platform Specific Component Development

The principle behind the PYRAMID approach is a separation of concerns by only dealing with the problems that are specific to the model abstraction level being considered at the time. The platform independent component development process defined the required component functionality without consideration for how it was to be achieved (see section 3); the next step of the component development process involves the consideration of the non-functional requirements and the infrastructure specification that have been defined in conjunction with the System Designer (see section 4).

As is generally important throughout the deployment process there should be regular sharing and communication between any different parties carrying out the different engineering roles. Decisions made during platform specific system design, such as the choice and establishment of system infrastructure architecture approaches or mechanisms, should be captured appropriately and discussed throughout component development. The method for this should enable all relevant information to be captured, and one option is to evolve a specification from the PIM component specifications. Additionally, a suitable approach for verification and integration of components on the platform specific system should be agreed. The contribution towards safety, security and other non-functional requirements provided by a component or the infrastructure should also be discussed and updated as development continues.

5.1 Infrastructure Considerations

5.1.1 Extensible Data

The Data Driving policy describes how data can be specified for operational use (e.g. to provide the weapon fit for a mission). Data can be delimited and/or encoded, which enables extension of the data without having a direct impact on components that are making use of the data.

Extensibility can be achieved by relating individual parts of the data with an identifier or tag along with an indication of the extent of the data; this allows components to identify which data the extension requires and ignore data that the extension does not require. Data can therefore be extended to support a particular component or extension component requirement without affecting existing interface integrations.

An example is the use of Session Description Protocol subunits within the Session Initiation Protocol (SIP), Ref. [40], to define supported capability. If the receiver supports the capability it can decipher the message; if it does not it can understand enough to say that capability is not supported.

When selecting a format for the data, care should be taken to consider whether the adoption of data representation standards may result in the need for safety certifiable data translation library code. Safety certifiable library code with appropriate verification artefacts may need to be supplied to all Component Developers to avoid duplication of effort on a deployment programme. Similarly, the security aspects of processing data need to be given appropriate attention.

5.1.2 Health Monitoring and Fault Management

During the platform independent component development phase described in section 3 it is assumed that the component will execute the expected behaviour, and all the supporting infrastructure and components can be fully relied on. In the platform specific phase enough is known about the infrastructure (e.g. the operating system, application software, hardware or managed equipment) to make judgments on what can and cannot be relied on, and what the development implications are.

Components should be able to identify how loss of capability that they depend on impacts their own capability. Documenting service dependencies, typically in a service dependencies diagram, could help indicate implications of failure in the chain.

Components will have to provide information that will be analysed by the health monitoring components; see the Health Management policy.

5.1.3 Interaction with Humans

Specific components are provided for interaction with humans; see the HMI policy. However these HMI components do not have any 'domain' knowledge, so to meet human interaction requirements other components will be involved via the HMI components, and the implications of this should be considered. For example, needs to present information, such as a communications heat map, would require additional data to be held within an appropriate component.

5.1.4 Interaction with Equipment

Components are needed to determine the capability of equipment or parts of the software infrastructure, and to understand how to control hardware resources. For guidance on developing these components see the Interaction with Equipment policy and Interfacing with Deployable Assets policy. Some guidance on the boundary of the PRA with respect to equipment hardware is also given in the 'Smart Equipment and the PRA-Equipment System Boundary' section of the Interaction with Equipment policy. Interface requirements with existing equipment also need to be taken into account.

5.2 Component Implementation

Given a set of requirements, there are many ways in which to realise a component; these are not covered at any length in this document. However, it is recommended that any code developed with a PRA deployment in mind should be data-driven (provided that it is done in a way that is feasible within the development time and cost constraints). This means that representative data sets should be provided to any System Integrator; see section 5.3.

5.2.1 Legacy Code Reuse

The PRA and this document cover the idea of design reuse primarily at the system level, with the goal of saving time and effort in a development programme. However this does not rule out code reuse within a PRA deployment. Below is a summary of considerations for code reuse based on the guidance in the FACE reference guide, Ref. [26], which covers similar issues:

- Determine the grouping and applicability of components for the legacy system:
    - Split the legacy system into multiple components or keep it as a single deployment?
    - Is the software still supportable?
    - Can the software be adapted to the PRA?
- Characterise and assess the legacy software functionality (Ref. [26] describes this functionality as software-based "capability"):
    - Was the functionality designed for capability monitoring, safety, or security?
    - What other functionalities does this code interface to?
    - What types of interfaces are there and what standards were used for those interfaces?
- Assess how the functionality fits into the PRA architecture:
    - Have component group boundaries been defined for the functionality?
    - To what protection domain(s) do the component group(s) belong?
    - Is a more restrictive profile applicable for future use of this functionality?
- Determine how to integrate the functionality into the architecture without polluting components:
    - Should the legacy architecture be converted to align to the PRA?
    - Should the legacy architecture interface to new PRA based system functionality hosted within the existing architecture?
    - Should the legacy architecture interface to new PRA based system functionality hosted within an architecture aligned to the PRA?

Where refactoring of legacy code would be required, the expected costs should first be evaluated against the benefits of refactoring the code. Reverification, capturing of safety artefacts, and capability update issues should be considered where applicable. Additionally, where legacy code from a standard product is being specialised or tailored during integration, the full impact of this should be carefully considered to weigh the benefits and issues, such as future support challenges arising from changes to the product.

5.2.2 Automatic Code Generation

The traditional approach to software development (see Figure 25) was to treat design and development as elaborative process steps, where "fidelity" was added that was considered 'too low level' for the analysis model, with independent assessment of completeness. This often led to vague and incomplete analysis models (Ref. [9]).

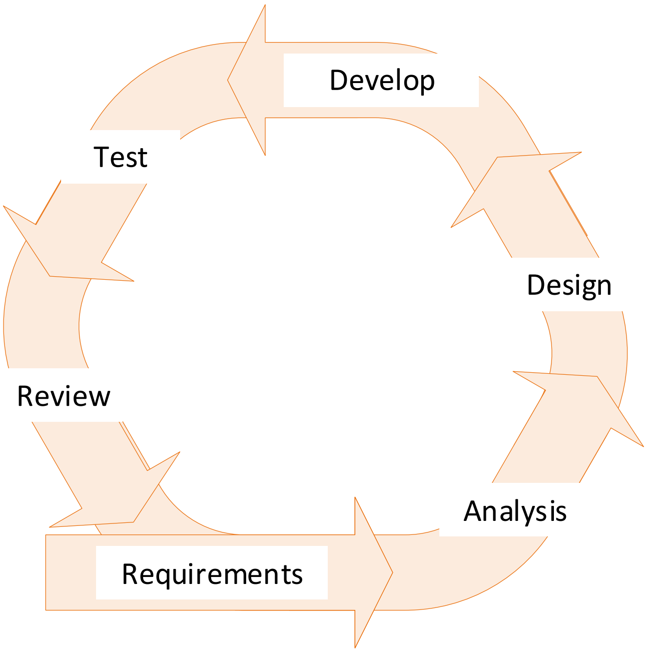


Figure 25: Traditional Software Development Lifecycle

By moving some of the rigour into the model it is possible to automate the generation of large parts of the code; this improves the consistency of code, facilitates documentation of behaviour (in the model), and, dependent on toolchain support, allows for faster rework. An implementation specific model can be produced to allow more technology specific targets to be built direct from a model.

When using auto-generation of code from a model, it is likely that any validation rules required for the resultant system still need to be applied, e.g. DO-326A, Ref. [21]. For details on how to use a Model Driven Architecture (MDA) approach to support a model based approach to software development see Ref. [9].

Note that executable code is not the only thing that can be generated from the model. Wherever possible, documentation should also be generated from the model, allowing the model to be the single source of truth. Data models and service interfaces can be used instead of, or to produce, interface control documents.

5.3 Working with System Integrators

Modelled components are by design not complete applications in their own right. Therefore individual components, or groups of components, will always require integrating into a larger deployment. A decision has to be made through discussion with the System Integrator on the component group to be delivered as a mini-deployment, and whether internally used services and interfaces are defined at a PIM or PSM level (even if not accessible in the executable software) to allow reuse of the modelled components in a different configuration.

To give examples of context for a component of interest, in order to determine potential broader use of the component in other deployments, Component Developers can use interaction views (Appendix C of Ref. [2]). System Integrators of other potential deployments should be engaged at the earliest opportunity.

5.4 Good Practice

This section provides good practice recommendations applicable to typical cases of platform specific component development, including guidance for avoiding common pitfalls.

5.4.1 Do consider appropriate middleware access

While a component will access the resources provided by or via middleware, this interaction must never be:

- Beyond the scope of the component's subject matter.
- Instead of using the component's defined interface.

A component can directly interact with middleware where it is within the scope of that component to do so, for instance directly accessing stored data when it is owned by the component; this is true no matter what role a component has within the control architecture.

5.4.2 Do manage safety artefact development

An agreement between Component Developers and System Designers on how appropriate safety artefact development will be managed should be reached early in the development process, so that dependencies can be identified. The agreed management methods should cover the mechanisms for provision of any applicable safety evidence from a Component Developer to the System Designer. Frequent information sharing between the parties is good practice, and is especially important regarding safety related developments.

5.5 Summary

This section covered additional considerations that help mature a component based on execution platform considerations. The expectation is that this will be an iterative process, and the development of the component will be done in parallel to the system design and integration discussed in section 4, the final goal being to generate executable software. The main points discussed in this section were:

- **Interaction with Humans**, consider components that capture services required to support the HMI interactions with the system, see section 5.1.3.
- **Component Implementation**, either through software development, code reuse or automatic code generation; see section 5.2.

Appendix A Component Connection Examples

Four component connection patterns are identified within the Component Connections policy in PYRAMID Exploiter's Pack Annex A, Ref. [2]. This appendix contains examples of the use of those patterns.

Appendix A.1 Coordination – Atomic Transaction

One of the most commonly used component interaction patterns is atomic transaction coordination. It is used when actions or data need to be synchronised between components that are not direct counterparts. For example, atomic transaction coordination can be used for resource allocation, when moving to new modes of operation, or for data loading.

This service pattern is only appropriate if the transaction passes the ACID test, which means that it is:

- **Atomic** – All participants, including any subordinate components, can be treated as a single 'atomic' unit, with all components changing together or all failing together.
- **Consistent** – None of the changes violate the validity of data in the components; if they do all the changes need to be rolled back.
- **Isolated** – Multiple instances of the service can occur concurrently, without interfering with each other.
- **Durable** – Upon successful completion the change will survive any subsequent failures of new transactions.

There are several protocol definitions for this service pattern, and other similar ACID transactions; one example is the Web Service Atomic Transaction standard, Ref. [57].

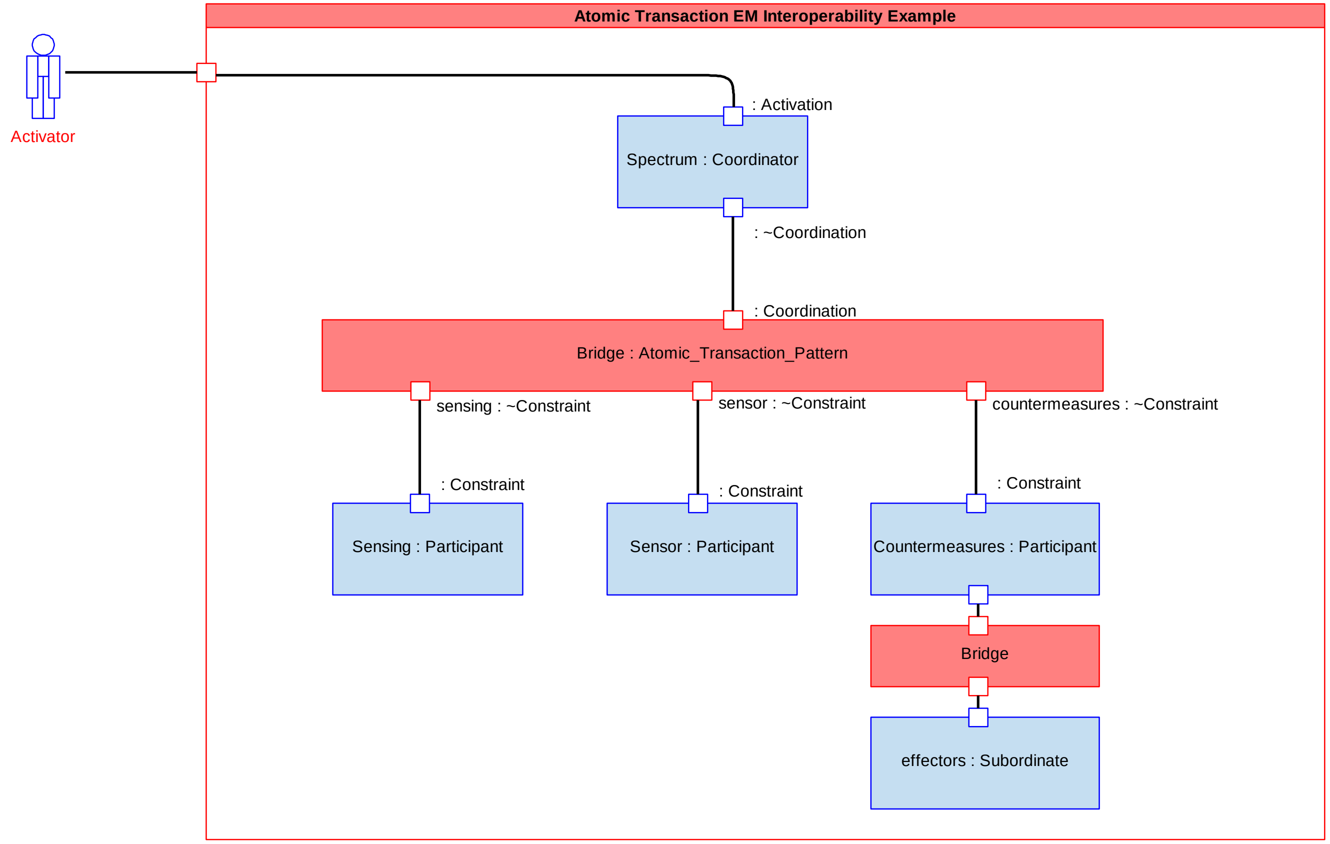
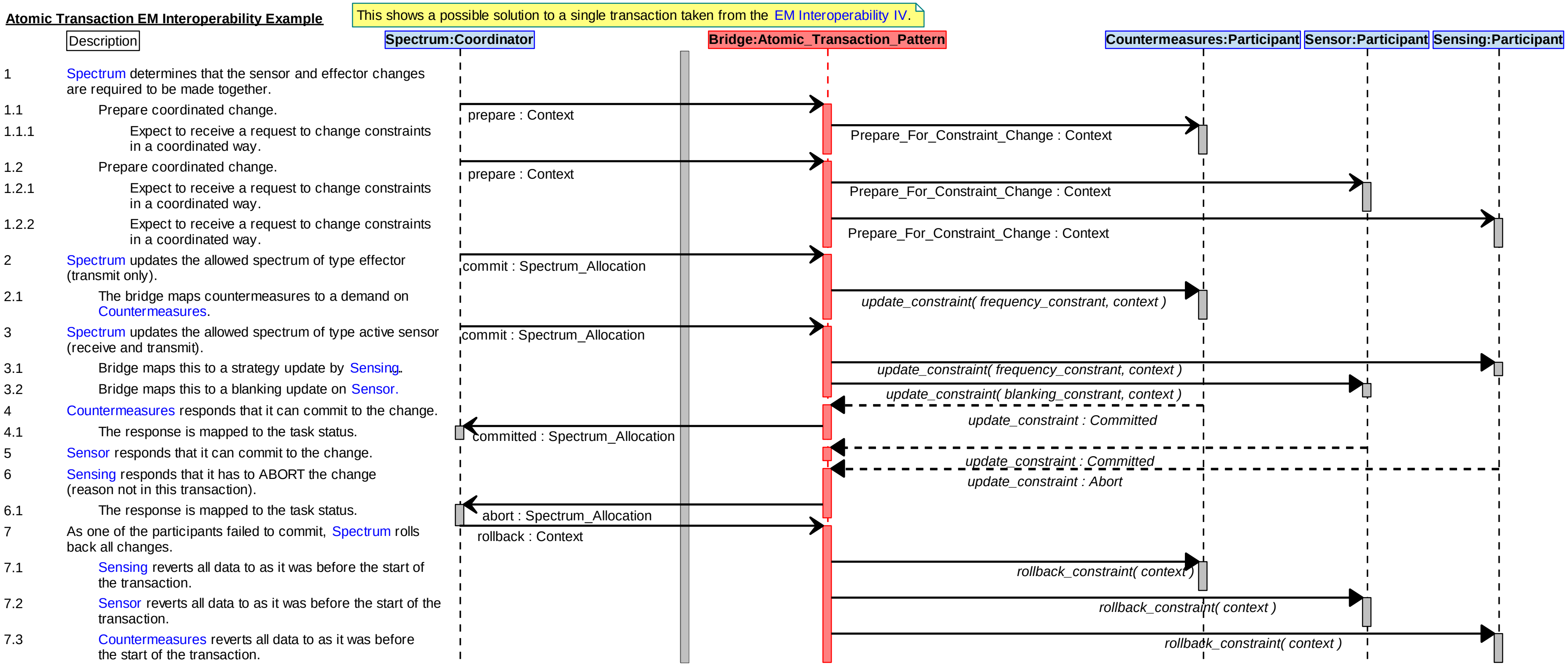


Figure 26: Atomic Transaction EM Interoperability

The example atomic transaction shown in Figure 26 is analogous to a single coordination from the EM Interoperability interaction view (IV) in the PYRAMID model. In this example three components (Sensing, Sensors and Countermeasures) have their frequency allocation updated in a coordinated way by the Spectrum component in (steps 7.1, 7.2 and 7.3 in Figure 27 below). The new frequency allocation is added as a change to the constraint on all the components; for more detail see the Constrain Capability use case in section B.1.2 of Ref. [2]. To avoid any conflicting allocation all components need to be updated together.

The details of an atomic transaction for EM operability can be seen in Figure 27.



Atomic Transaction EM Interoperability Example

This shows a possible solution to a single transaction taken from the EM Interoperability IV.

Description

1 Spectrum determines that the sensor and effector changes are required to be made together.
1.1 Prepare coordinated change.
1.1.1 Expect to receive a request to change constraints in a coordinated way.
1.2 Prepare coordinated change.
1.2.1 Expect to receive a request to change constraints in a coordinated way.
1.2.2 Expect to receive a request to change constraints in a coordinated way.
2 Spectrum updates the allowed spectrum of type effector (transmit only).
2.1 The bridge maps countermeasures to a demand on Countermeasures.
3 Spectrum updates the allowed spectrum of type active sensor (receive and transmit).
3.1 Bridge maps this to a strategy update by Sensing.
3.2 Bridge maps this to a blanking update on Sensor.
4 Countermeasures responds that it can commit to the change.
4.1 The response is mapped to the task status.
5 Sensor responds that it can commit to the change.
6 Sensing responds that it has to ABORT the change (reason not in this transaction).
6.1 The response is mapped to the task status.
7 As one of the participants failed to commit, Spectrum rolls back all changes.
7.1 Sensing reverts all data to as it was before the start of the transaction.
7.2 Sensor reverts all data to as it was before the start of the transaction.
7.3 Countermeasures reverts all data to as it was before the start of the transaction.

Spectrum:Coordinator
Bridge:Atomic_Transaction_Pattern
Countermeasures:Participant
Sensor:Participant
Sensing:Participant
prepare : Context
Prepare_For_Constraint_Change : Context
prepare : Context
Prepare_For_Constraint_Change : Context
Prepare_For_Constraint_Change : Context
commit : Spectrum_Allocation
update_constraint(frequency_constrant, context)
commit : Spectrum_Allocation
update_constraint(frequency_constrant, context)
update_constraint(blanking_constrant, context)
update_constraint : Committed
committed : Spectrum_Allocation
update_constraint : Committed
update_constraint : Abort
abort : Spectrum_Allocation
rollback : Context
rollback_constraint(context)
rollback_constraint(context)
rollback_constraint(context)

Figure 27: Atomic Transaction EM Interoperability Sequence Diagram

First Spectrum **prepares** those involved for the coordinated change. As components are agnostic of other components, with requests based on their own understanding, it is possible that there is not always a one-to-one match between coordination requests and component services; this is managed as a mapping in the bridge. In this example the bridge duplicates the sensor request to Sensing and Sensors components. The prepared request includes the type of change (but not the details), allowing components to also prepare subordinate components for the change. They can do this using a simple transaction, another atomic transaction (often using the same ID) or a counterpart relationship.

The second stage of the transaction is a request for all components to **commit** to the details of a change, within the provided timeout. If any of the components fail to respond within the timeout, or as in this example one of the components responds that it has to abort, then Spectrum request a rollback from all involved, resetting all the constraints to how they were before the transaction started. The resolution to why the transaction was aborted is the subject of a different set of transactions.

In this example all the properties changed are of the same type, but that does not have to be the case, as long as the change passes the ACID test. This kind of transaction is often used even with single participants, allowing them to prepare subordinate components for the change. However the isolation requirement means that this coordination type is only really appropriate to small actions or data changes; for complex actions that require coordination, an alternative approach is required, such as the Business Agreement covered in the next section.

Appendix A.2 Coordination – Business Agreement

The second common component interaction pattern is Business Activity coordination. This is used when the priorities of multiple parts of the system need to be coordinated when coming up with a solution. For example, as far as the Sensors component is concerned, the best search solution may be to turn on all active sensors, whereas this is the worst solution as far as the Observability component is concerned; a middle ground that both are happy with needs to be chosen.

The coordinator's aim is to reach an agreement on a level of service based around a set of derived requirements, with the solution owned by the participants.

As with the Atomic Transaction coordination there are several standards for this service pattern; one example is the Web Service Business Activity standard, Ref. [58].

The example Business Activity Coordination pattern shown in Figure 28 is a snapshot of a single coordination from the Jettison Management IV, where Jettison asks Asset Transitions to determine a solution that will make the jettison package releasable (i.e. opening doors before release), and requests Stores Release to provide a stores jettison solution.

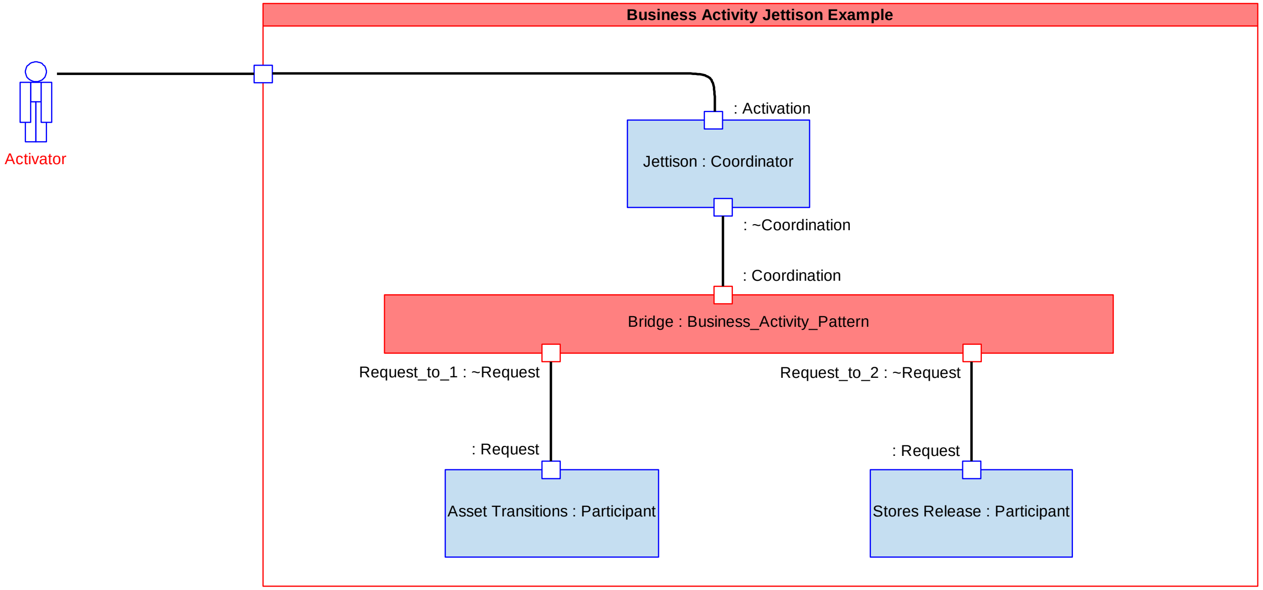
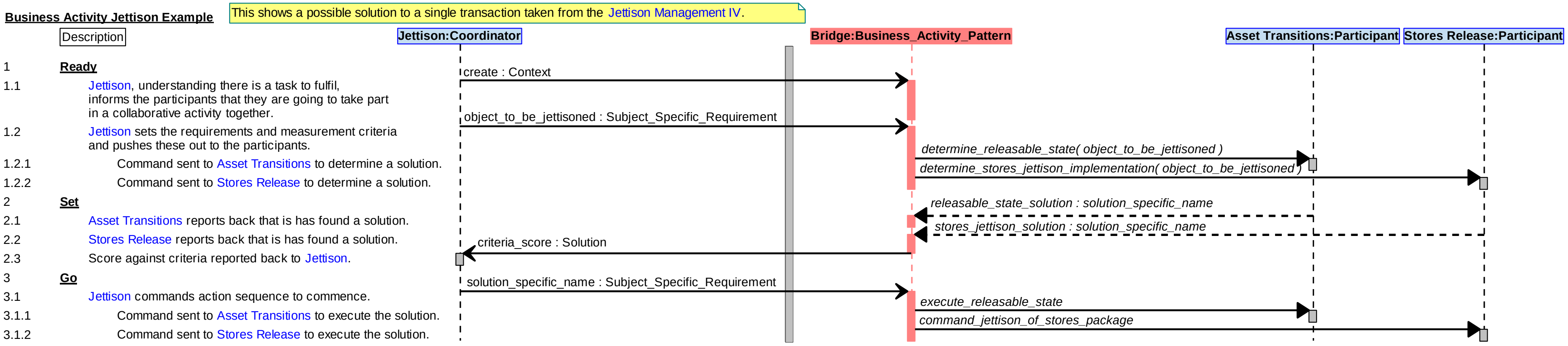


Figure 28: Business Activity Jettison Example

As can be seen in Figure 29, this component interaction pattern begins with requirements being placed on the two components from Jettison. The requirements take the form of one or more criteria and a measurement of success. All participants determine a proposed solution against their requirements, and provide scores against the criteria.



Business Activity Jettison Example

This shows a possible solution to a single transaction taken from the Jettison Management IV.

Description

1 Ready
1.1 Jettison, understanding there is a task to fulfil, informs the participants that they are going to take part in a collaborative activity together.
1.2 Jettison sets the requirements and measurement criteria and pushes these out to the participants.
1.2.1 Command sent to Asset Transitions to determine a solution.
1.2.2 Command sent to Stores Release to determine a solution.
2 Set
2.1 Asset Transitions reports back that is has found a solution.
2.2 Stores Release reports back that is has found a solution.
2.3 Score against criteria reported back to Jettison.
3 Go
3.1 Jettison commands action sequence to commence.
3.1.1 Command sent to Asset Transitions to execute the solution.
3.1.2 Command sent to Stores Release to execute the solution.

Jettison:Coordinator
Bridge:Business_Activity_Pattern
Asset Transitions:Participant
Stores Release:Participant
create : Context
object_to_be_jettisoned : Subject_Specific_Requirement
determine_releasable_state(object_to_be_jettisoned)
determine_stores_jettison_implementation(object_to_be_jettisoned)
releasable_state_solution : solution_specific_name
stores_jettison_solution : solution_specific_name
criteria_score : Solution
solution_specific_name : Subject_Specific_Requirement
execute_releasable_state
command_jettison_of_stores_package

Figure 29: Business Activity Sequence Diagram Jettison Example

The second stage is the coordinator selecting the solution to enact. As the actual details of the solution are based on the subject matter of participant components (not the coordinator), the coordinator selects the best solution based on the score of the solutions and the score of any pre-requisites required for the solution. In this case the criterion provided by Jettison is binary (you can jettison or not), but for other transactions this is likely to be a more nuanced measure, for example including time or resource cost. Once a solution is selected by the coordinator, the participants are informed of the choice and when the solution is to be enacted (at an agreed time or on demand).

The final stage of the pattern is for the participants to enact the solution. Jettison can monitor progress of achievement against the requirement. Asset Transitions and Stores Release are best placed to determine whether their own requirements have been met, and the coordinator determines completion of the overall requirement.

Note: If for any reason the service level agreed against the requirement can no longer be met, then the coordinator is to be informed; however, if the requirement can still be met by a new solution without impacting other components, this is within the scope of the participants to select.

Business Activity Coordination can often be for very long term and high level goals. This leads to a further derivation of requirements within the deployment as part of the top level requirement (as shown in Figure 30), with each subject matter area owning its own solution, and derived requirements.

This coordination differs from an Atomic Transaction in three key ways:

- The coordinator only owns derivation of the requirements; the participant determines how the solution is achieved, allowing it to be used at an abstract level.
- Business Activity transactions involve requirement derivation and solution determination, followed by solution selection and then a final enactment stage, whereas the steps in an Atomic Transaction are just for the components to prepare and then commit.
- There is no requirement to be able to roll-back to a previous state, allowing it to be used for long term irreversible change such as ‘fly to this location’.

It should be noted that the pattern for solution derivation does not require 'coordination'. As in the Business Activity Jettison Example above, if the derived requirements and solutions are independent of each other they can be treated as separate transactions. The ‘keeping track’ of the dependencies can then be considered ‘invisible’ outside of the coordinating component. Taken to its logical extreme this stops becoming so much of a coordination between participants and more a directed series of steps, and this approach is covered in the orchestration pattern in the next section.

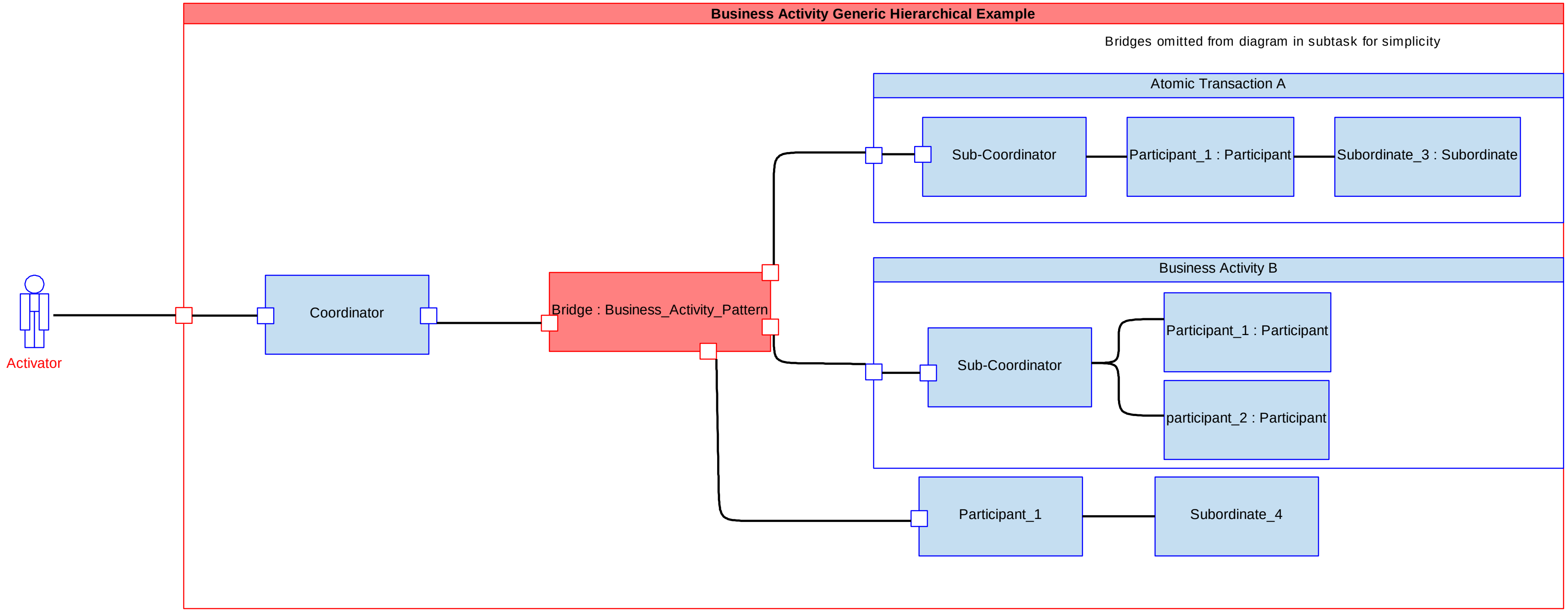

Business Activity Generic Hierarchical Example
Bridges omitted from diagram in subtask for simplicity
Atomic Transaction A
Sub-Coordinator
Participant_1 : Participant
Subordinate_3 : Subordinate
Business Activity B
Sub-Coordinator
Participant_1 : Participant
participant_2 : Participant
Activator
Coordinator
Bridge : Business_Activity_Pattern
Participant_1
Subordinate_4

Figure 30: Business Activity Generic Hierarchical Example

Appendix A.3 Orchestration

The Orchestration component interaction pattern differs from a coordination transaction in that the orchestrator owns the workflow logic, rather than just the requirement derivation. This can be a very versatile approach, where multiple building block components can be reused in a variety of different configurations and orders to achieve different results.

The process itself is usually data-driven, allowing a lot of flexibility, with multiple standards existing for the process execution; one example is the Web Service Business Process Execution Language standard, Ref. [59].

The example Orchestration component interaction pattern (Figure 31) shows Asset Transitions in a role that is similar to the role it plays in the Startup IV. In this interaction view, Asset Transitions manages the workflow of starting up an air vehicle across multiple interactions, driving out the process steps one by one to achieve an overall result. This pattern involves orchestrating multiple component interaction patterns across multiple services, either in parallel or in series depending on the workflow dependencies. This example involves three stages all activated by the same request.

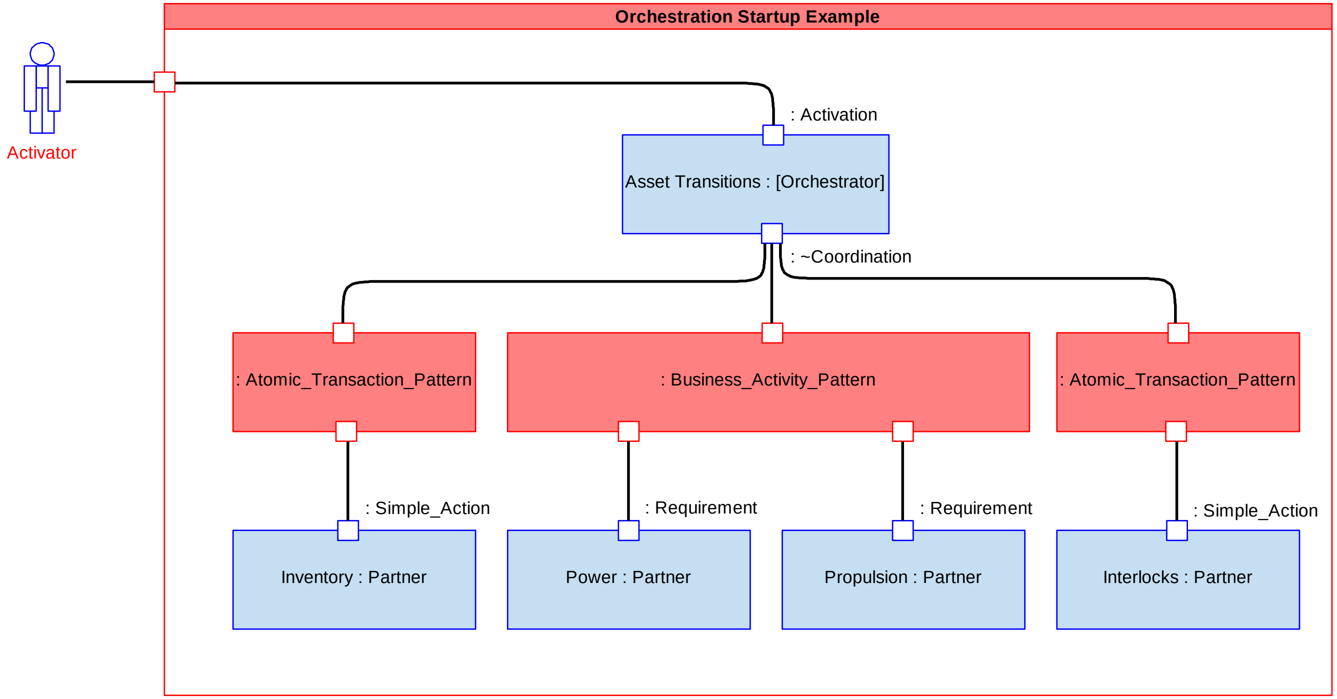
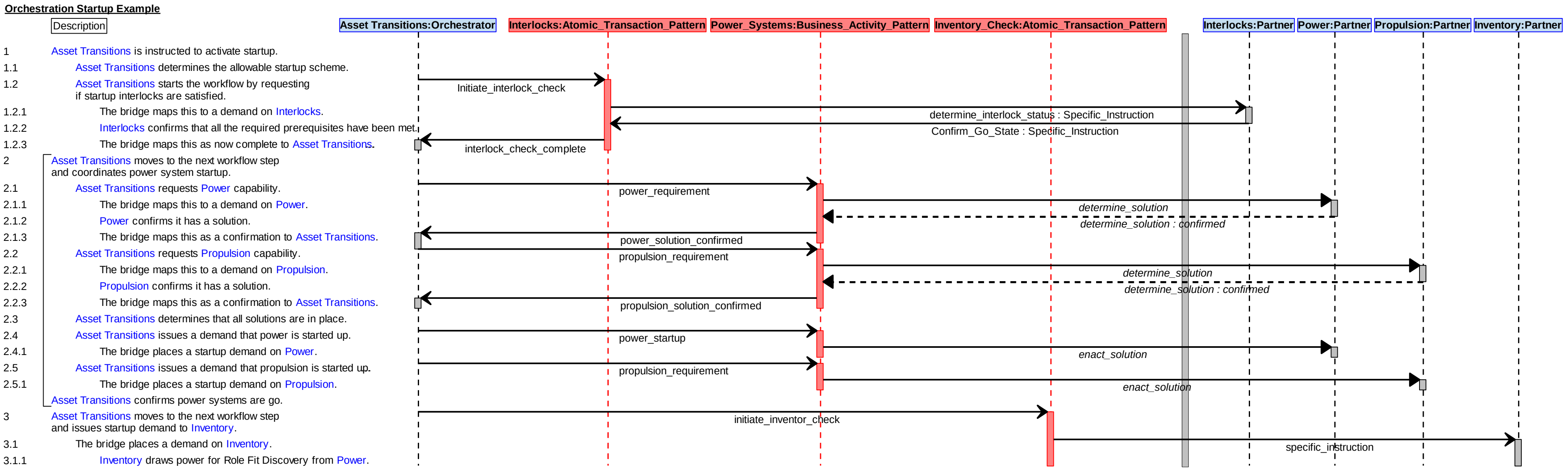


Figure 31: Orchestration Startup Example Diagram

As can be seen in Figure 32, this component interaction pattern begins with Asset Transitions determining the correct sequence of steps and dependencies. The first step is to check with Interlocks that the correct authorisations and interlocks have been fulfilled to allow startup. Asset Transitions doesn't need to know the details of what is required, just whether permission is granted or not.

When this task is complete Asset Transitions requests that Power and Propulsion both determine if they can bring basic check power online at a given time. This is done as a business activity coordination, because power and propulsion systems may be able to start up independently but not together. Again Asset Transitions does not need to know the details (they are outside its subject matter area), just whether a solution for initial power is in place. As a solution exists, application of limited power to a store attachment mechanism can be started.

The last step of the process flow shown is the triggering of Inventory to do a stores fit check. In reality there will be further checks but this illustrates the pattern.



Orchestration Startup Example

Description

1 Asset Transitions is instructed to activate startup.

1.1 Asset Transitions determines the allowable startup scheme.

1.2 Asset Transitions starts the workflow by requesting if startup interlocks are satisfied.

1.2.1 The bridge maps this to a demand on Interlocks.

1.2.2 Interlocks confirms that all the required prerequisites have been met.

1.2.3 The bridge maps this as now complete to Asset Transitions.

2 Asset Transitions moves to the next workflow step and coordinates power system startup.

2.1 Asset Transitions requests Power capability.

2.1.1 The bridge maps this to a demand on Power.

2.1.2 Power confirms it has a solution.

2.1.3 The bridge maps this as a confirmation to Asset Transitions.

2.2 Asset Transitions requests Propulsion capability.

2.2.1 The bridge maps this to a demand on Propulsion.

2.2.2 Propulsion confirms it has a solution.

2.2.3 The bridge maps this as a confirmation to Asset Transitions.

2.3 Asset Transitions determines that all solutions are in place.

2.4 Asset Transitions issues a demand that power is started up.

2.4.1 The bridge places a startup demand on Power.

2.5 Asset Transitions issues a demand that propulsion is started up.

2.5.1 The bridge places a startup demand on Propulsion.

Asset Transitions confirms power systems are go.

3 Asset Transitions moves to the next workflow step and issues startup demand to Inventory.

3.1 The bridge places a demand on Inventory.

3.1.1 Inventory draws power for Role Fit Discovery from Power.

Asset Transitions:Orchestrator
Interlocks:Atomic_Transaction_Pattern
Power_Systems:Business_Activity_Pattern
Inventory_Check:Atomic_Transaction_Pattern
Interlocks:Partner
Power:Partner
Propulsion:Partner
Inventory:Partner
Initiate_interlock_check
determine_interlock_status : Specific_Instruction
Confirm_Go_State : Specific_Instruction
interlock_check_complete
power_requirement
determine_solution
determine_solution : confirmed
power_solution_confirmed
propulsion_requirement
determine_solution
determine_solution : confirmed
propulsion_solution_confirmed
power_startup
enact_solution
propulsion_requirement
enact_solution
initiate_inventor_check
specific_instruction

Figure 32: Orchestration Startup Example Sequence Diagram

Appendix A.4 Choreography

An important characteristic of the choreography interaction pattern is that it is about collaboration, not control. This interaction pattern is useful where the goal is to establish the rules and limits for interactions between collaborating components. Choreographies are often used between very disparate groups of components that operate autonomously and have not explicitly been designed to work together, for example an on-ground system of radars and facilities that may occasionally interact with air vehicles. They can operate at relatively specific levels of detail; for example informing components of which counterpart relationships to setup when a new store is attached to the air vehicle (e.g. different components are involved depending on whether it is a fuel tank or a missile).

The component that owns the choreography and the collaborating components need to be able to identify the context of the collaboration before the choreography transaction itself starts. Data loaded in the components from common or at least aligned mission plans enables the collaborators to register for involvement in the particular collaboration group, typically when a set of conditions are met (e.g. this may simply be at startup).

The transaction is initiated by the collaborators registering with the choreographer, and requesting the rules and limits. The collaborators then use the information supplied as the basis for direct interactions, as shown in Figure 33. The choreographer knows about the different groups involved, but is unlikely to control their actions.

As you might expect, the details of the rules can be quite complex, and there are several protocols designed explicitly to express this. See Ref. [56] for a comparison of three of the standards.

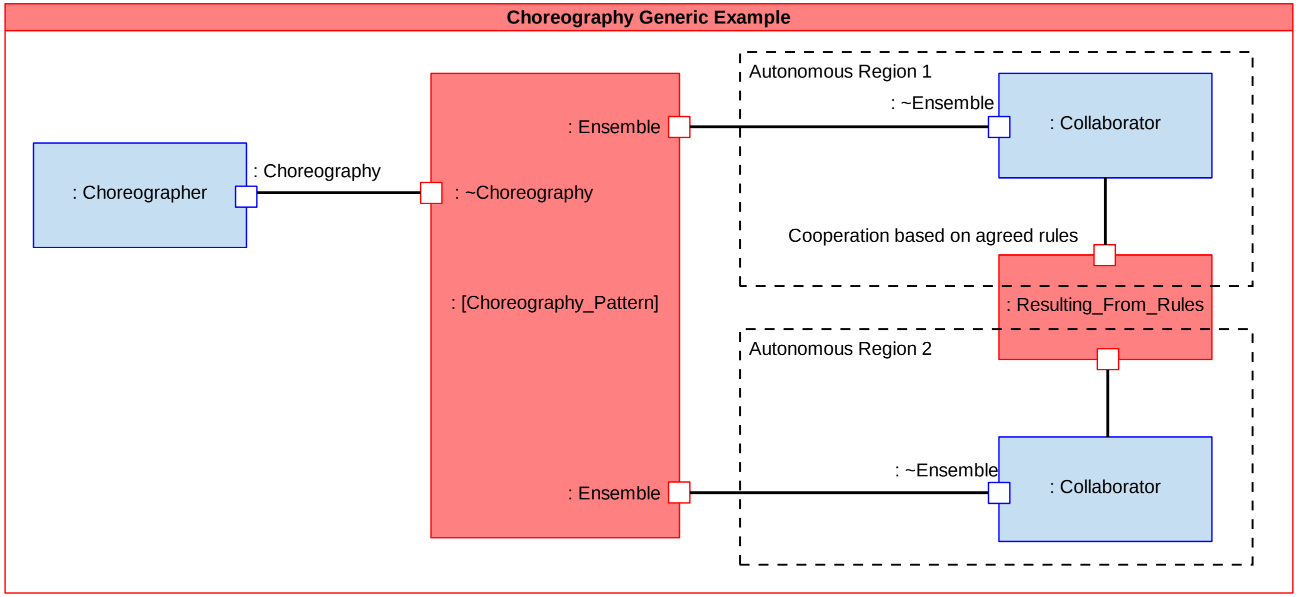


Figure 33: Choreography Generic Example

In the example shown in Figure 34 and Figure 35, the air platform is required to perform an air-to-air refuelling (AAR). The Formations component can be used to set up a collaboration relationship between the Routes and Tactical Objects components, so as to enable Formations to coordinate a suitable rendezvous for refuelling. In this scenario, the tanker aircraft is not a PRA based platform and cannot accept, nor provide, manoeuvring control.

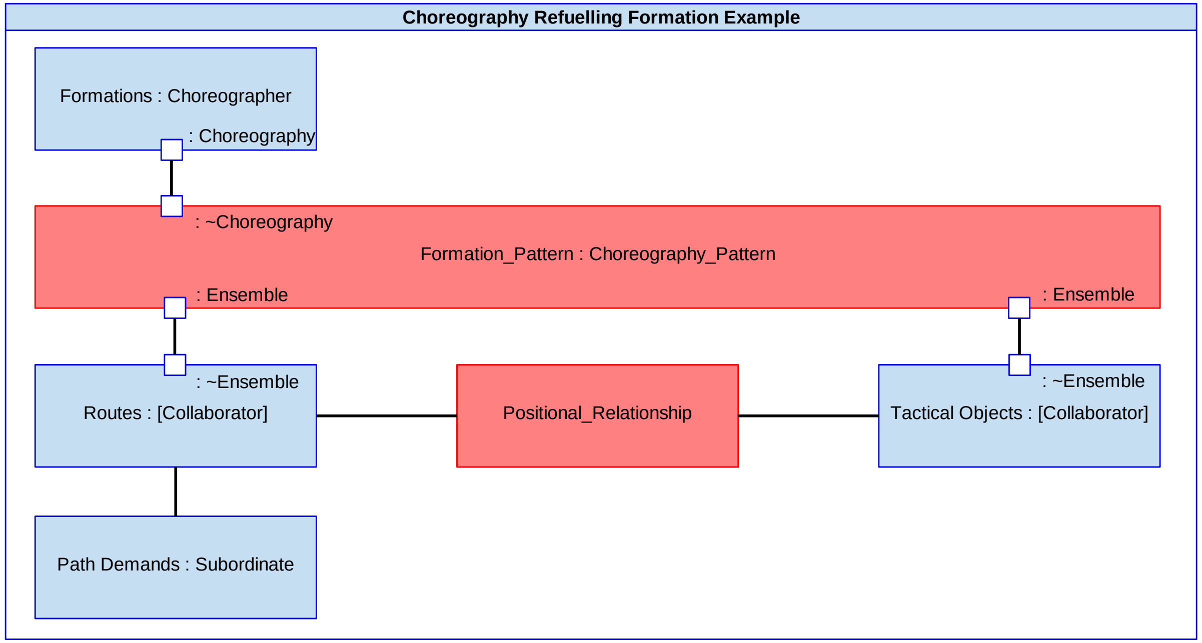
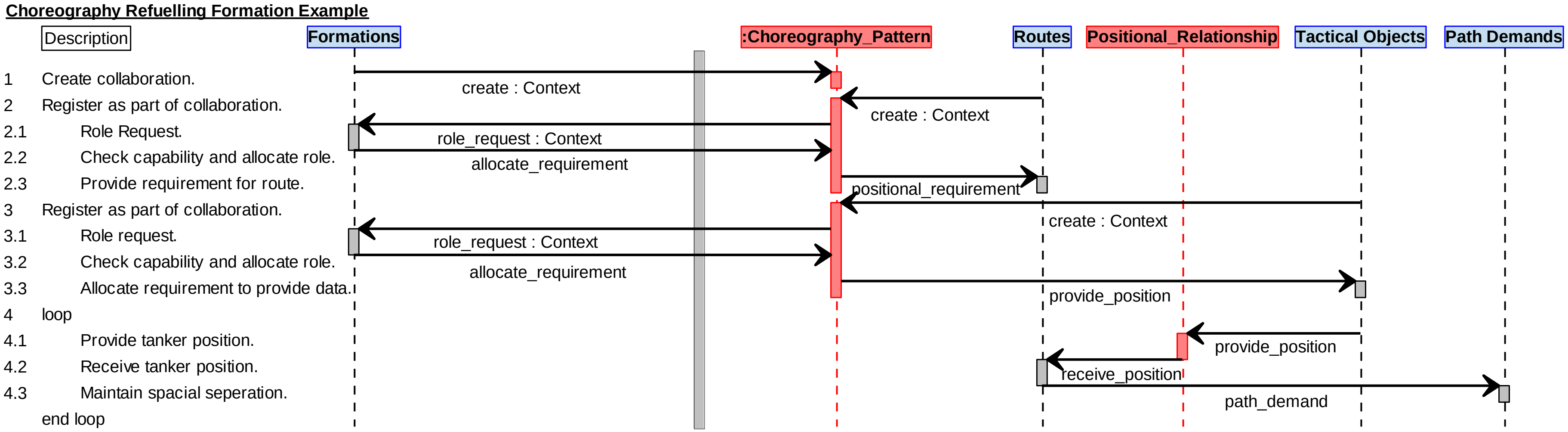

Choreography Refuelling Formation Example
Formations : Choreographer
: Choreography
: ~Choreography
Formation_Pattern : Choreography_Pattern
: Ensemble
: Ensemble
: ~Ensemble
Routes : [Collaborator]
Positional_Relationship
: ~Ensemble
Tactical Objects : [Collaborator]
Path Demands : Subordinate

Figure 34: Choreography Refuelling Formation Example

The Formations component can provide the initial positional offset that the air vehicle should adopt before joining formation, and it is provided with an identifier for the tanker aircraft, but not its current position. Therefore two services are required, one that can route the air vehicle to a specified location, and another that can provide the position of the tanker aircraft to which a positional offset can be applied.

The Tactical Objects component has the capability to provide the tanker position, and the Routes component has the capability to direct the manoeuvring of the aircraft. These capabilities are reported to the Formations component via the bridge, which then allocates the service provider based on their stated capabilities.

Since the Formations component does not need to know the actual position of the tanker, the pattern can be used to instruct Tactical Objects to transfer the positional information directly to the Routes component. The Routes component can then use this information to generate a route to the rendezvous point and request the manoeuvres via commands to Path Demands.



Choreography Refuelling Formation Example

Description

1 Create collaboration.
2 Register as part of collaboration.
2.1 Role Request.
2.2 Check capability and allocate role.
2.3 Provide requirement for route.
3 Register as part of collaboration.
3.1 Role request.
3.2 Check capability and allocate role.
3.3 Allocate requirement to provide data.
4 loop
4.1 Provide tanker position.
4.2 Receive tanker position.
4.3 Maintain spacial seperation.
end loop

Figure 35: Choreography Refuelling Formation Example Sequence

Roles are allocated so that the required services are provided to Formations, and its consumption of services associated with the other components involved in the choreography is monitored. So long as the components have the capability to provide the required services, they can be left to manoeuvre the aircraft without further input from the Formations component.

Once the air vehicle reaches the location of the air tanker, a new positioning requirement is generated and this relationship can be removed.