Project: COMPASS
Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

Final Report on SoS Architectural Models

Document Number: D22.6
Date: September 2014

Public Document

http://www.compass-research.eu
Contributors:
Simon Perry (SAP), ATEGO
Jon Holt (JDH), ATEGO
Richard Payne (RP), UNUT
Jeremy Bryans (JB), UNUT
Claire Ingram (CI), UNUT
Alvaro Miyazawa (AM), UY
Stefan Hallerstede (SH), AU
Luis D Couto (LC), AU
Anders Kaels Malmos (AKM), AU
Juliano Iyoda (JI), UFPE
Marcio Cornelio (MC), UFPE
Jan Peleska (JP), UBRE

Editors:
Simon Perry, ATEGO

Reviewers:
Luis D Couto, AU
Uwe Schulze, UB
Jim Woodcock, UY
Samuel Canham, UY
## Document History

### Release Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2014-09-08</td>
<td>S. A. Perry</td>
<td>First complete version for internal review</td>
</tr>
<tr>
<td>1.1</td>
<td>2014-09-26</td>
<td>S. A. Perry</td>
<td>Updated with some review comments.</td>
</tr>
<tr>
<td>1.2</td>
<td>2014-09-29</td>
<td>R. J. Payne</td>
<td>Updated to address Contract Patterns review comments</td>
</tr>
</tbody>
</table>
Abstract

This deliverable contains the final report on modelling patterns and architectures for system of systems (SoSs) and their constituent systems (CSs). Fundamental architectural principles for systems and SoSs are discussed and summarised and a range of different types of SoS are classified and related to a number of example patterns that are applicable to SoS engineering. These patterns are catalogued and described and are categorised as either architectural patterns used to define SoS architectures or enabling patterns used to enable aspects of the modelling of an architecture, such as interfaces, evolution through time, testing, traceability etc.

Executive Summary

Introduction

This deliverable presents an overview of the fundamental definitions and principals associated with architectures and architectural frameworks and presents a collection of modelling patterns to be used when engineering Systems of Systems. An executive summary of these main areas is given in the following sections.

Fundamental Definitions and Principals

The concept of an architecture is fundamental to any systems engineering undertaking. There is much confusion, however, due to the plethora of very similar terms involving the word ‘architecture’. The use of architecture has also changed significantly over the years as the definitions that apply to the world of software engineering have a significantly-simpler scope than the world of systems engineering. A related concept is that of the architectural framework. This deliverable has used a number of sources to abstract the fundamental definition and principals relating to architectures and architectural frameworks, as follows. For full details see Section 2.

Abstraction of Definitions

The following ontology covering the concepts related to architecture and architectural frameworks is abstracted from the information sources considered and defines the terms and concepts that are used in this document.
The concepts shown on Figure 1 are defined as follows:

- **Architectural Framework** - a defined set of Viewpoints and an Ontology. The Architectural Framework is used to structure an Architecture from the point of view of a specific industry, stakeholder role set, or organisation. The Architectural Framework is defined so that it meets the needs defined in its Architectural Framework Context.

- **Architectural Framework Context** - defines the needs (also known in standards such as ISO42010 as concerns) that an Architectural Framework has to address.

- **Ontology** - an element of an Architectural Framework that defines all the concepts and terms (Ontology Elements) that relate to any Architecture structured according to the Architectural Framework.

- **Ontology Element** - the concepts that make up an Ontology. Ontology Elements can be related to each other and are used in the definition of each Viewpoint (through the Viewpoint Elements that make up a Viewpoint).

- **Viewpoint** - a definition of the structure and content of a View. The content and structure of a Viewpoint uses the concepts and terms from the Ontology via the Viewpoint Elements that make up the Viewpoint. Each Viewpoint is defined so that it meets the needs defined in its Viewpoint Context.

- **Viewpoint Context** - defines the needs (also known in standards such as ISO42010 as concerns) that a Viewpoint has to address.
- **Viewpoint Element** - the elements that make up a Viewpoint. Each Viewpoint Element must correspond to an Ontology Element from the Ontology that is part of the Architectural Framework.

- **Architecture** - a description of a System, made up of a number of Views. Related Views can be collected together into Perspectives.

- **View** - the visualisation of part of the Architecture of a System, that conforms to the structure and content defined in a Viewpoint. A View is made up of a number of View Elements.

- **View Element** - the elements that make up a View. Each View Element visualises a Viewpoint Element that makes up the Viewpoint to which the View, on which the View Element appears, conforms.

- **Perspective** - a collection of Views (and hence also their defining Viewpoints) that are related by their purpose. That is, Views which address the same architectural needs, rather than being related in some other way, such as by mode of visualisation, for example.

- **Rule** - a construct that constrains the Architectural Framework (and hence the resulting Architecture) in some way, for example by defining minimum required Viewpoints.

- **System** - set of interacting elements organised to satisfy one or more needs. The artefact being engineered that the Architecture describes.

It is important to note here that an architecture is simply considered to be a description of a system, represented by a number of views that are created according to a number of predefined viewpoints from a given architectural framework.

The architectural framework that is used in the production of a system architecture may be an existing framework (such as MODAF or TRAK) or may be a framework created specifically for a particular project. For guidance of creating a bespoke architectural framework, see “Definition of the COMPASS Architectural Framework Framework” [D21.5b 2014].

**Abstraction of Principles**

The following key principles relating to the development of architectures can be identified:

- Architectures are core to systems and SoS engineering; production of architectures is not an optional activity.

- Architecting takes place throughout the system life cycle and resulting architectures must evolve over time.

- Architectures should be produced according to defined architectural viewpoints codified in an architectural framework that includes consistency rules defined between the various views produced and the information contained in them.
Architectures should be produced to address the concerns of stakeholders using relevant architectural viewpoints; it is important to know why a particular viewpoint is being used.

It is essential that an architectural design process is defined and followed. This must cover the definition, analysis, evaluation, documentation and maintenance of an architecture.

Multiple architectures may be needed (such as: functional, physical and system architectures). These should be produced at a particular level of abstraction.

Architectures should address both structure and behaviour, including communications, functionality and data flow.

Mapping between architectures should be done at the same level of abstraction.

Modelling is essential to the development of architectures.

Architectural styles and patterns should be used whenever possible, as should open systems and loose coupling. Architectural reuse should be encouraged.

When used for a SoS, architectures should support design by contract, and dynamic reconfiguration of constituent systems.

When used for a SoS, architecture is not about the constituent systems, but about how they work together.

Modelling Patterns

The main part of this deliverable presents a collection of modelling patterns that can be used as part of a model-based approach to systems engineering. For full details see Section 4.

The Two Types of Pattern Considered

This deliverable considers two types of modelling pattern:

1. Architectural patterns – Architectural patterns describe specific system architectures, both in terms of structure and behaviour, which address particular needs of the system. An example of an architectural pattern would be one that addresses a particular type of control for a system (e.g. centralised vs. distributed).

2. Enabling patterns – Enabling patterns are specific constructs of modelling elements whose combination and subsequent use enables a number of systems engineering applications. An example of an enabling pattern would be one used for the definition of interfaces or one used to ensure traceability throughout a model of a system. While not used explicitly to define the structure and behaviour of an architecture, they are used to
enable aspects of the modelling of an architecture, such as interfaces, evolution through time, testing, traceability etc.

It should be noted here that the use of system covers both the concepts of System of Systems (SoSs) and the Constituent Systems (CSs) that make up an SoS.

**Architectural Patterns**

This deliverable considers eight architectural patterns:

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>AIMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralised</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Centralised control and management of SoS</td>
</tr>
<tr>
<td></td>
<td>• Reuse of pre-existing systems</td>
</tr>
<tr>
<td>Service Oriented</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Analysis of SoS emergent behaviour</td>
</tr>
<tr>
<td></td>
<td>• SoS/constituent system evolution</td>
</tr>
<tr>
<td></td>
<td>• Central SoS authority</td>
</tr>
<tr>
<td></td>
<td>• Enable cross-domain SoS development</td>
</tr>
<tr>
<td></td>
<td>• Long SoS lifecycle</td>
</tr>
<tr>
<td>Publish-Subscribe</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Loose coupling between publisher and subscriber</td>
</tr>
<tr>
<td></td>
<td>• constituent systems in the SoS with time, flow and space</td>
</tr>
<tr>
<td></td>
<td>• decoupling between the constituent systems</td>
</tr>
<tr>
<td></td>
<td>• One to many and many to many communications between the constituent systems in the SoS</td>
</tr>
<tr>
<td>Pipes and Filters</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Data or material flow oriented systems</td>
</tr>
<tr>
<td></td>
<td>• Independent processing steps on a flow</td>
</tr>
<tr>
<td></td>
<td>• Configurable transmission of the flow between processing elements</td>
</tr>
<tr>
<td></td>
<td>• Dynamic change of processing steps and connectors</td>
</tr>
<tr>
<td>Supply Chain</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Efficient and fast product manufacture and delivery across large distances</td>
</tr>
<tr>
<td></td>
<td>• Collaboration between suppliers and integrators</td>
</tr>
<tr>
<td></td>
<td>• Easily configurable chains, allowing chains to adapt to</td>
</tr>
<tr>
<td></td>
<td>• new opportunities or to problems that arise</td>
</tr>
<tr>
<td>Blackboard</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td></td>
<td>• Development of expert or knowledge based systems</td>
</tr>
<tr>
<td></td>
<td>• Loose coupling</td>
</tr>
<tr>
<td></td>
<td>• Separation of concerns</td>
</tr>
<tr>
<td>Infrastructure Grid</td>
<td>The main aims of this pattern are to support:</td>
</tr>
<tr>
<td>Grid</td>
<td>• Allowing the flow of energy/material around a large area</td>
</tr>
<tr>
<td></td>
<td>• Providing planning and management of the flow of</td>
</tr>
<tr>
<td></td>
<td>• material/energy on a very large scale</td>
</tr>
<tr>
<td></td>
<td>• Retaining autonomy within distinct geographic areas</td>
</tr>
<tr>
<td></td>
<td>• Allowing market competition</td>
</tr>
</tbody>
</table>
The main aims of this pattern are to:

- Allow the SoS to adapt when a CS ceases to provide a function (at some minimum level of performance) by adopting an alternative provider
- Ensure that the situation where architectural reconfiguration is required can be identified
- Ensure that architectural alternatives can be evaluated and the best choice selected

It is not our expectation that an SoS will exhibit one pattern only. Most SoSs are capable of implementing multiple patterns at once. For example, a Publish-Subscribe pattern may be employed along with a Reconfiguration Control Architecture.

**Enabling Patterns**

This deliverable considers six enabling patterns:

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>AIMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Definition Pattern</td>
<td>The aim of this pattern is identification of interfaces and their relation to the system elements that use them and the ports that expose them. This includes:</td>
</tr>
<tr>
<td></td>
<td>• Defining the interfaces in terms of the operations they may provide.</td>
</tr>
<tr>
<td></td>
<td>• Defining the interfaces in terms of the flows of data, material, energy, personnel etc. that take place across an interface.</td>
</tr>
<tr>
<td></td>
<td>• Identification of the connections between ports that expose interfaces and of the interface connections that take place across those port connections.</td>
</tr>
<tr>
<td></td>
<td>• Definition of any protocols to which an interface or port must conform.</td>
</tr>
<tr>
<td></td>
<td>• Identification of typical scenarios showing how interfaces are used.</td>
</tr>
<tr>
<td>PATTERN</td>
<td>AIMS</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Test Pattern</td>
<td>The aim of this pattern is to define tests, which may be broken down into the following:</td>
</tr>
<tr>
<td></td>
<td>• Definition of testing context, to allow the type of testing, level of testing, testing constraints and any necessary system to be identified and defined by considering the testing context.</td>
</tr>
<tr>
<td></td>
<td>• Definition of test set-up, to allow the testing context to be satisfied by defining an overall testing schedule.</td>
</tr>
<tr>
<td></td>
<td>• Definition of test cases, to define the individual test cases that allow all aspects of the system-under-test to be tested.</td>
</tr>
<tr>
<td>Traceability Pattern</td>
<td>The aim of this pattern is to allow a systems engineer to establish traceability in a model of a system, constrained by the need to:</td>
</tr>
<tr>
<td></td>
<td>• Support the capture of the traceability relationships between traceable elements in a systems engineering model, throughout the model and between any desired model elements.</td>
</tr>
<tr>
<td></td>
<td>• Support systems engineers and systems engineering managers in the identification of model elements that may be impacted by change.</td>
</tr>
<tr>
<td></td>
<td>When defining allowed traceability, the pattern:</td>
</tr>
<tr>
<td></td>
<td>• Supports the definition of the types of traces that can be used.</td>
</tr>
<tr>
<td></td>
<td>Supports the definition of the types of elements that can be involved in trace relationships and the relationships that can be used between such traceable elements.</td>
</tr>
<tr>
<td>Life Cycles Pattern</td>
<td>The aim of this pattern is to allow a systems engineer and a systems engineering manager to understand system life cycles, constrained by the need to apply to any type of life cycle, including (but not limited to):</td>
</tr>
<tr>
<td></td>
<td>• Project life cycles</td>
</tr>
<tr>
<td></td>
<td>• Product life cycles</td>
</tr>
<tr>
<td></td>
<td>• Procurement life cycles</td>
</tr>
<tr>
<td></td>
<td>The pattern allows life cycles to be understood both in terms of their structure and behaviour, with the ability to understand the interactions between life cycles, allowing multiple life cycles for a system to be related one to another.</td>
</tr>
<tr>
<td>PATTERN</td>
<td>AIMS</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>Epoch (System evolution) Pattern</td>
<td>The aim of this pattern is to understand the evolution of a system by considering the evolution of the complexity of the system over time. It allows epochs to be defined, the key viewpoints to be considered in each epoch to be identified and metrics used to measure the evolution to be defined.</td>
</tr>
<tr>
<td>Contract Pattern</td>
<td>The aim of this pattern is to enable the analysis of SoS emergent behaviour, by modelling a SoS in terms of contracts. The patterns supports:</td>
</tr>
<tr>
<td></td>
<td>• Identification of contracts in an SoS, where the SoS engineer determines the various contracts required to provide the emergent behaviours of the SoS, which must restrict the behaviours acceptable for a CS to join the SoS.</td>
</tr>
<tr>
<td></td>
<td>• Identification of conformance of constituent systems to contracts, to ensure that the CSs of a SoS conform to contracts defined by the SoS engineer and so that the engineer is able to dictate the conformance relations in the SoS.</td>
</tr>
<tr>
<td></td>
<td>• Identification of connections between contracts, to enable the SoS engineer to identify how the contracted behaviours should be linked in an SoS.</td>
</tr>
<tr>
<td></td>
<td>• Definition of contract functionality – defining the contracts in terms of the state and operations they provide to their environment and internal operations.</td>
</tr>
<tr>
<td></td>
<td>• Definition of the ordering of external communications and internal state – definition of any protocols to which a contract must conform.</td>
</tr>
</tbody>
</table>

### Future Steps

The enabling patterns work has proved to be particularly fruitful, being used within the COMPASS project as inputs to various frameworks as well as the B&O case study. See “Final Report on Guidelines for SoS Engineering” [D21.6 2014] for further details. They are also being used widely by Atego with a number of their customers on real engineering projects. In addition, one of the authors is continuing to investigate new enabling patterns as part of INCOSE UK’s Model-Based Systems Engineering Working Group.

As well as continuing to investigate candidate patterns, further research is needed on the relationship between patterns and the CML. Rather than the current approach of transformation of SysML into CML based purely on the underlying SysML element (such as a block or a state), further research is needed into how such transformations can recognise and be dependent on the patterns that are being used in a SysML model.
# Table of Contents

**DOCUMENT HISTORY** .............................................................................................................3

**ABSTRACT** ...............................................................................................................................4

**EXECUTIVE SUMMARY** .........................................................................................................4

**TABLE OF CONTENTS** ..........................................................................................................12

**FIGURES** ................................................................................................................................16

**TABLES** ....................................................................................................................................22

## 1. INTRODUCTION ..................................................................................................................23

1.1. **SCOPE** ............................................................................................................................23

1.2. **CONTEXT** ..........................................................................................................................23

1.3. **DOCUMENT STRUCTURE** ...............................................................................................24

## 2. DISCUSSION OF ARCHITECTURAL PRINCIPLES FOR SOS ....................................26

2.1. **FUNDAMENTAL DEFINITIONS** .......................................................................................26

2.1.1. **CONCEPTS FROM SYSTEMS AND SOFTWARE ENGINEERING LITERATURE** ..........26

2.1.2. **CONCEPTS AS DESCRIBED BY THE INTERNATIONAL STANDARDS ORGANISATION (ISO)** ....27

2.1.3. **CONCEPTS AS DESCRIBED BY THE INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING (INCOSE)** .....28

2.1.4. **CONCEPTS AS DESCRIBED BY THE ARCHITECTURAL FRAMEWORK COMMUNITY** ......29

2.1.4.1. **THE ZACHMAN FRAMEWORK** ..............................................................................29

2.1.4.2. **DEFENCE-BASED ARCHITECTURE FRAMEWORKS** ............................................29

2.1.4.3. **NON-DEFENCE ARCHITECTURE FRAMEWORKS** ...............................................31

2.2. **EXISTING SYSTEMS-LEVEL ARCHITECTURAL TECHNIQUES** ..................................32

2.2.1. **MODEL-DRIVEN ARCHITECTURE** ...........................................................................32

2.2.2. **ABSTRACTION FROM DICKERSON & MAVRIS "ARCHITECTURE AND PRINCIPLES OF SYSTEMS ENGINEERING"** .................................................................34

2.2.3. **ISO42010** ....................................................................................................................37

2.2.4. **ISO15288 ARCHITECTURAL DESIGN PROCESS** ....................................................38

2.2.5. **ACADEMIC search** ....................................................................................................40

2.3. **EXISTING SYSTEM OF SYSTEMS-LEVEL ARCHITECTURAL TECHNIQUES** ...............45

2.3.1. **DoD ‘SYSTEMS ENGINEERING GUIDE FOR SYSTEMS OF SYSTEMS’** .................45

2.3.2. **ACADEMIC search** ....................................................................................................48

2.4. **ABSTRACTION OF DEFINITIONS AND PRINCIPLES** .............................................56

2.4.1. **ABSTRACTION OF DEFINITIONS** ............................................................................56

2.4.2. **ABSTRACTION OF PRINCIPLES** ............................................................................58

2.4.2.1. **FROM MDA** .........................................................................................................58

2.4.2.2. **FROM DICKERSON & MAVRIS** ............................................................................58

2.4.2.3. **ISO 42010** .............................................................................................................58

2.4.2.4. **ISO 15288** .............................................................................................................58

2.4.2.5. **FROM DoD ‘SYSTEMS ENGINEERING GUIDE FOR SYSTEMS OF SYSTEMS’** ....59

2.4.2.6. **ACADEMIA** ..........................................................................................................59

2.5. **SUMMARY OF PRINCIPLES** ..........................................................................................59

## 3. CLASSIFICATION OF DIFFERENT TYPES OF SOS ..................................................61
3.1. SoS CHARACTERISTICS .......................................................... 61
3.2. SoS APPLICATION EXAMPLES ............................................... 65
3.2.1. COMPASS SoS CASE STUDIES ........................................... 65
3.2.1.1. BANG & OLUFSEN CASE STUDY .................................. 65
3.2.1.2. INSEL CASE STUDY ...................................................... 67
3.2.1.3. LONDON EMERGENCY SERVICES ................................... 68
3.2.2. INDUSTRIAL CASES ......................................................... 70
3.2.2.1. HEALTH CARE SYSTEMS ............................................. 70
3.2.2.2. FINANCE AND INVESTMENT SYSTEMS .............................. 72
3.2.2.3. PACKAGE SHIPPING AND TRACKING SYSTEMS ................. 73
3.2.2.4. MARITIME TRANSPORTATION SYSTEMS .......................... 74
3.2.2.5. SPACE EXPLORATION SYSTEMS ..................................... 75
3.2.2.6. METEOROLOGY SYSTEMS ............................................. 76
3.2.2.7. AIR TRAFFIC CONTROL SYSTEMS ................................. 78
3.2.2.8. DEFENCE SYSTEMS ..................................................... 79
3.2.2.9. RECONFIGURABLE UNMANNED AIRCRAFT SYSTEMS ............ 80
3.2.3. EXAMPLES FROM COMPASS INTEREST GROUP ..................... 81
3.2.3.1. SMART GRID SYSTEMS ................................................ 81
3.2.3.2. INTEGRATED MODULAR AVIONICS ................................ 83
3.2.3.3. ROAD TRAFFIC MANAGEMENT ...................................... 84
3.3. RELATIONS TO SoS ARCHITECTURAL PATTERNS ...................... 85
3.4. SUMMARY ............................................................................ 86

4. PATTERNS COLLECTION ................................................................ 87

4.1. WHAT ARE MODELLING PATTERNS? .......................................... 87
4.2. DEFINING PATTERNS ............................................................... 88
4.2.1. DESCRIBING ARCHITECTURAL PATTERNS ............................ 89
4.2.2. DESCRIBING ENABLING PATTERNS .................................... 90
4.3. ARCHITECTURAL PATTERNS .................................................. 92
4.3.1. CENTRALISED ARCHITECTURE PATTERN ............................ 93
4.3.1.1. INTRODUCTION .......................................................... 93
4.3.1.2. SOLUTION ................................................................. 94
4.3.1.3. SUMMARY ................................................................. 98
4.3.2. SERVICE ORIENTED ARCHITECTURE .................................. 100
4.3.2.1. INTRODUCTION .......................................................... 100
4.3.2.2. SOLUTION ................................................................. 101
4.3.2.3. SUMMARY ................................................................. 106
4.3.3. PUBLISH-SUBSCRIBE ARCHITECTURE PATTERN .................. 107
4.3.3.1. INTRODUCTION .......................................................... 107
4.3.3.2. SOLUTION ................................................................. 110
4.3.3.3. SUMMARY ................................................................. 113
4.3.4. PIPES AND FILTERS ARCHITECTURE PATTERN .................... 114
4.3.4.1. INTRODUCTION .......................................................... 114
4.3.4.2. SOLUTION ................................................................. 115
4.3.4.3. SUMMARY ................................................................. 118
4.3.5. SUPPLY CHAIN PATTERN .................................................... 119
4.3.5.1. INTRODUCTION .......................................................... 119
4.3.5.2. SOLUTION ................................................................. 121
4.3.5.3. SUMMARY ................................................................. 123
D22.6 – Final Report on SoS Architectural Models (Public)

4.4.4.5. **LIFE CYCLE VIEWPOINT (LCV)** ................................................................. 232
4.4.4.6. **LIFE CYCLE MODEL VIEWPOINT (LCMV)** .............................................. 234
4.4.4.7. **INTERACTION IDENTIFICATION VIEWPOINT (IIV)** ................................. 236
4.4.4.8. **INTERACTION BEHAVIOUR VIEWPOINT (IBV)** ....................................... 238
4.4.4.9. **EXTENSIONS TO THE LIFE CYCLE PATTERN** ........................................ 241
4.4.4.10. **SUMMARY** ......................................................................................... 241
4.4.5. **EPOCH PATTERN** .................................................................................... 242
4.4.5.1. **PATTERN AIMS** .................................................................................... 242
4.4.5.2. **CONCEPTS** .......................................................................................... 243
4.4.5.3. **PATTERN VIEWPOINTS** .......................................................................... 244
4.4.5.4. **RULES** ................................................................................................ 245
4.4.5.5. **EPOCH DEFINITION VIEWPOINT (EDV)** .................................................. 247
4.4.5.6. **APPLICABLE VIEWSET VIEWPOINT (AVV)** ............................................. 249
4.4.5.7. **METRIC DEFINITION VIEWPOINT (MDV)** ............................................... 251
4.4.5.8. **METRIC USAGE VIEWPOINT (MUV)** ...................................................... 255
4.4.5.9. **SUMMARY** ......................................................................................... 257
4.4.6. **CONTRACT PATTERN** ............................................................................. 259
4.4.6.1. **PATTERN AIMS** .................................................................................... 259
4.4.6.2. **CONCEPTS** .......................................................................................... 260
4.4.6.3. **PATTERN VIEWPOINTS** .......................................................................... 261
4.4.6.4. **RULES** ................................................................................................ 262
4.4.6.5. **CONTRACTUAL SOS DEFINITION VIEWPOINT (CSDV)** ......................... 262
4.4.6.6. **CONTRACT CONFORMANCE VIEWPOINT (CCV)** ..................................... 264
4.4.6.7. **CONTRACT CONNECTIONS VIEWPOINT (CConnV)** ................................... 266
4.4.6.8. **CONTRACT DEFINITION VIEWPOINT (CDV)** .......................................... 268
4.4.6.9. **CONTRACT PROTOCOL DEFINITION VIEWPOINT (CPV)** ....................... 272

5. **SUMMARY** .................................................................................................... 275

6. **REFERENCES** ............................................................................................... 276

**APPENDIX A - AUTOMATED MODEL-BASED TESTING (MBT) AND THE TEST PATTERN** ................................................................. 288

A.1. **CASE STUDY: TURN INDICATION FUNCTION** .............................................. 288
A.2. **RT-Tester Test Modelling Approach** ............................................................ 289
A.3. **TESTING CONTEXT VIEW** ........................................................................... 289
A.4. **TEST BEHAVIOUR VIEW** ............................................................................ 291
A.5. **TEST CASE GENERATION** .......................................................................... 294
A.6. **AUTOMATED TRACEABILITY DATA GENERATION** ....................................... 294
A.7. **TEST PROCEDURES, TEST RECORDS AND TEST SCHEDULES** ................ 296

**APPENDIX B - REFINEMENT IN SYSML** .......................................................... 298

B.1. **FORMAL SEMANTICS OF SysML MODELS** ................................................ 298
B.2. **GUIDELINES OF USAGE** ............................................................................. 300
B.3. **REFINEMENT IN SysML** ............................................................................. 301
B.4. **SysML EXTENSIONS** .................................................................................. 302
B.5. **REFINEMENT LAWS IN SysML** ................................................................. 307
Figures

Figure 1 - Ontology for Architectures and Architectural Frameworks........................................5
Figure 2 - Summary of the ISO 42010 architecture-related concepts........................................28
Figure 3 - Summary of the MODAF architecture-related concepts...........................................30
Figure 4 - Summary of the TRAK architecture-related concepts..............................................32
Figure 5 - Overview of MDA....................................................................................................33
Figure 6 - ISO 15288 Architectural Design Process ...................................................................39
Figure 7 - Ontology for Architectures and Architectural Frameworks........................................56
Figure 8 - Base B&O SoS........................................................................................................66
Figure 9 - Insitel Case Study architectural topology.................................................................68
Figure 10 - Emergency Services Case Study Topology............................................................69
Figure 11 – National Health Care SoS.......................................................................................70
Figure 12 – A scenario with a constituent system collaboration with two systems of systems.................................................................................................................................71
Figure 13 – Finance and Investment Sub-systems ......................................................................73
Figure 14 – Package Tracking System of Systems ....................................................................73
Figure 15 – Example of a complex sub-system.........................................................................74
Figure 16 – SSMN Network topology crop................................................................................76
Figure 17 - GCOS Overview.....................................................................................................77
Figure 18 – ATMP Free Flight for air traffic control...................................................................79
Figure 19 - SysML Model for ATMP........................................................................................79
Figure 20 – Example of a Digitized Battlefield .......................................................................80
Figure 21 – Data Oriented Architecture for future UAS System of Systems.............................81
Figure 22 – GRID Architecture Overview................................................................................82
Figure 23 - IMA layers [Prisaznuk92].......................................................................................84
Figure 24 - The Composite Pattern............................................................................................88
Figure 24 - Typical System Hierarchy Showing Stereotypes from the Composite Pattern.........................................................................................................................................................89
Figure 26 - Elements of an architectural pattern........................................................................90
Figure 27 - Centralised Architectural Pattern Ontology............................................................94
Figure 28 - SoS with strict centralised architecture....................................................................95
Figure 29 - SoS with hybrid centralised-decentralised architecture..........................................96
Figure 30 - Centralised SoS example - Military command example [Hall-May&06].................97
Figure 31 - Centralised SoS example - Transport management SoS........................................98
Figure 32 - Service Oriented Architecture Architectural Pattern Ontology...........................100
Figure 33 - Illustration of Services in a Component-Based System.......................................102
Figure 34 - Travel Agent SoS example - SysML internal block diagram.................................104
Figure 35 - Constituent system composition of travel agent services.....................................105
Figure 36 - Event-Based Publish-Subscribe Conceptual model..............................................108
Figure 37 - Simplified Data-Centric Publish-Subscribe Conceptual Model [OMG-DDS 2007]..................................................................................................................................................................................108
Figure 38 - Complete Data-Centric Publish-Subscribe Conceptual Model [OMG-DDS 2007]..............................................................................................................................................................................109
Figure 39 - Example of a SoS with two domains.................................................................110
Figure 40 – Data-Centric Publish Subscribe Medicine Card SoS Example.............................112
Figure 41 - Data-Centric Publish Subscribe Package Tracking SoS Example.............112
Figure 42 - Pipes and Filters Architecture Pattern Conceptual Model.................................114
Figure 43 - SoS Pipes and Filter pattern .................................................................115
Figure 44 - Modified SoS Pipes and Filter Pattern with control & configuration .................................................................116
Figure 45 – Modified SoS Pipes and Filter Pattern with more inputs or outputs ..............................................................................116
Figure 46 - Software oriented SoS Pipes and Filter Example ..............................................117
Figure 47 - Material oriented SoS..................................................................................118
Figure 48 Example of supplier, integrator and logistics CSs in a supply chain SoS ..............................................................................120
Figure 49 - Blackboard Architecture Pattern Conceptual Model ........................................124
Figure 50 – Blackboard pattern with possible attributes and operations.............125
Figure 51 - SoS Blackboard Architecture Pattern .........................................................126
Figure 52 – SoS Healthcare Blackboard Example ..............................................................127
Figure 53 - SoS-RadarSat Blackboard Example .................................................................128
Figure 54 Decentralised Grid Infrastructure Pattern .....................................................132
Figure 55 Partially Centralised Grid Infrastructure Pattern ..............................................132
Figure 56 Data-Centralised Grid Infrastructure Pattern ..................................................132
Figure 57 RCA Sub-types. Uncontrolled RCA pattern (left); centralised RCA pattern (centre); and de-centralised RCA (right)...............137
Figure 58 - Architectural Framework Context View showing Interface Definition aims ..............................................................................142
Figure 59 – Ontology Definition View showing Interface Definition concepts...143
Figure 60 – Viewpoint Relationships View showing Interface Definition Viewpoints ..............................................................................144
Figure 61 - Rules Definition View showing Interface Definition rules .................145
Figure 62 - Viewpoint Context View showing Interface Identification Viewpoint aims ......................................................................................146
Figure 63 – Viewpoint Definition View showing the elements that appear on the Interface Identification Viewpoint (IIV) ........................................................................146
Figure 64 - An example of an Interface Identification View showing Service-Based Interfaces ........................................................................147
Figure 65 - An example of an Interface Identification View showing Flow-Based Interfaces ............................................................................148
Figure 66 - Viewpoint Context View showing Interface Connectivity Viewpoint aims ......................................................................................149
Figure 67 - Viewpoint Definition View showing the elements that appear on the Interface Connectivity Viewpoint (ICV) .............................................................150
Figure 68 - An example of an Interface Connectivity View showing Service-Based Interfaces ............................................................................151
Figure 69 - An example of an Interface Connectivity View showing Flow-Based Interfaces ..............................................................................152
Figure 70 - Viewpoint Context View showing Interface Definition Viewpoint aims ......................................................................................153
Figure 71 - Viewpoint Definition View showing the elements that appear on the Interface Definition View (IDV)........................................................................154
Figure 72 - An example of an Interface Definition View.................................................................................................................................155
Figure 73 - Viewpoint Context View showing Interface Behaviour Viewpoint aims ......................................................................................157
Figure 74 - Viewpoint Definition View showing the elements that appear on the Interface Behaviour View (IBV) ............................................................................................................ 157
Figure 75 - An example of an Interface Behaviour View showing Service-Based Interfaces .............................................................................................................. 158
Figure 76 - An example of an Interface Behaviour View showing Service-Based Interfaces with Interface explicitly shown .................................................................................. 159
Figure 77 - An example of an Interface Behaviour View showing Non-Service-Based Interfaces ........................................................................................................... 160
Figure 78 - Viewpoint Context View showing Protocol Definition Viewpoint aims ................................................................................................................................. 161
Figure 79 - Viewpoint Definition View showing the elements that appear on the Protocol Definition View (PDV) ................................................................................................. 162
Figure 80 - An example of a Protocol Definition View ............................................................................................................................................................................... 163
Figure 81 - An example of an Interface Behaviour View showing timing .............................................................................................................................................. 165
Figure 82 - An example of security limits applied to Port and Interface Connections .......................................................................................................................... 167
Figure 83 - An example of security limits applied to Flow Types used by an Interface ........................................................................................................................................ 168
Figure 84 - An example showing security limits applied to actual Flow Types ................................................................................................................................. 168
Figure 85 - An example showing defined and actual security limits for message parameters ..................................................................................................................................... 169
Figure 86 - An example of the definition of non-functional types .......................................................................................................................... 171
Figure 87 - An example of definition of sets of non-functional properties ................................................................................................................................. 172
Figure 88 - An example of an Interface Definition View showing associated non-functional properties ........................................................................................................ 173
Figure 89 - An example of an Interface Definition View using design by contract concepts .................................................................................................................................... 174
Figure 90 Architecture Framework Context View for the Test Pattern ........................................................................................................................................ 178
Figure 91 Ontology Definition View showing Test Pattern concepts .......................................................................................................................................... 179
Figure 92 Viewpoint Relationships View showing Test Pattern Viewpoints ..................................................................................................................................... 181
Figure 93 Rules Definition View showing Test Pattern rules .................................................................................................................................................. 182
Figure 94 Viewpoint Context View showing Testing Context Viewpoint ................................................................................................................................. 182
Figure 95 Viewpoint Definition View showing the elements that appear on the Testing Context Viewpoint ........................................................................................................ 183
Figure 96 An example of an testing Context View .................................................................................................................................................................................. 184
Figure 97 An example of an testing Context View defining additional validation constraints ................................................................................................................................... 185
Figure 98 Viewpoint Context View showing test Set-up Context Viewpoint ........................................................................................................................................ 186
Figure 99 Viewpoint Definition View showing the elements that appear on the test Set-up Viewpoint .................................................................................................................................. 187
Figure 100 - Example 'Test Structure View' .................................................................................................................................................................................. 188
Figure 101 Example 'Test Schedule Behaviour View' ............................................................................................................................................................................. 189
Figure 102 Example 'Test Set Behaviour View' ............................................................................................................................................................................... 190
Figure 103 Viewpoint Context View showing Test Case Viewpoint ........................................................................................................................................ 191
Figure 104 Viewpoint Context View showing Test Case Viewpoint ........................................................................................................................................ 192
Figure 105 Example 'Test Configuration View' showing the stakeholders ..................................................................................................................................... 194
Figure 106 Example 'Test behaviour View' showing a stakeholder-level scenario ........................................................................................................................................ 195
Figure 107 Satisfying the original 'Testing Context View' ............................................................................................................................................................................... 195
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>Example 'Test Configuration View' for processes</td>
<td>196</td>
</tr>
<tr>
<td>109</td>
<td>Example 'Test Behaviour View' for stakeholder-level scenario for a successful stunt scenario</td>
<td>196</td>
</tr>
<tr>
<td>110</td>
<td>Example 'Test Behaviour View' for stakeholder-level scenario for a failed stunt scenario</td>
<td>197</td>
</tr>
<tr>
<td>111</td>
<td>Satisfying use cases form the 'Testing Context View'</td>
<td>197</td>
</tr>
<tr>
<td>112</td>
<td>Example 'Test Configuration View' for system</td>
<td>198</td>
</tr>
<tr>
<td>113</td>
<td>Example 'Test Behaviour View' for system-level scenario for normal operation</td>
<td>198</td>
</tr>
<tr>
<td>114</td>
<td>Satisfying use cases form the 'Testing Context View'</td>
<td>199</td>
</tr>
<tr>
<td>115</td>
<td>Example 'Test Configuration View' for single system element</td>
<td>199</td>
</tr>
<tr>
<td>116</td>
<td>Example 'Test Behaviour View' for a single system element</td>
<td>200</td>
</tr>
<tr>
<td>117</td>
<td>Satisfying use cases form the 'Testing Context View'</td>
<td>200</td>
</tr>
<tr>
<td>118</td>
<td>Example 'Test Configuration View' for single process</td>
<td>201</td>
</tr>
<tr>
<td>119</td>
<td>Example 'Test Behaviour View' for a single process</td>
<td>201</td>
</tr>
<tr>
<td>120</td>
<td>Satisfying use cases form the 'Testing Context View'</td>
<td>202</td>
</tr>
<tr>
<td>121</td>
<td>Example 'Test Configuration View' for testing coverage of the model</td>
<td>202</td>
</tr>
<tr>
<td>122</td>
<td>Example 'Test Behaviour View' executing a testing process</td>
<td>203</td>
</tr>
<tr>
<td>123</td>
<td>Satisfying use cases form the 'Testing Context View'</td>
<td>203</td>
</tr>
<tr>
<td>124</td>
<td>Example 'Test Configuration View' for parametric values</td>
<td>204</td>
</tr>
<tr>
<td>125</td>
<td>Example 'Test Behaviour View' for system constraints</td>
<td>205</td>
</tr>
<tr>
<td>126</td>
<td>Architectural Framework Context View showing Traceability aims</td>
<td>208</td>
</tr>
<tr>
<td>127</td>
<td>Ontology Definition View showing Traceability concepts</td>
<td>209</td>
</tr>
<tr>
<td>128</td>
<td>Viewpoint Relationships View showing Traceability Viewpoints</td>
<td>210</td>
</tr>
<tr>
<td>129</td>
<td>Rules Definition View showing Traceability rules</td>
<td>212</td>
</tr>
<tr>
<td>130</td>
<td>Viewpoint Context View showing Relationship Identification Viewpoint aims</td>
<td>213</td>
</tr>
<tr>
<td>131</td>
<td>Viewpoint Definition View showing the elements that appear on the Relationship Identification View (RIV)</td>
<td>213</td>
</tr>
<tr>
<td>132</td>
<td>An example of a Relationship Identification View</td>
<td>214</td>
</tr>
<tr>
<td>133</td>
<td>Viewpoint Context View showing Traceability Identification Viewpoint aims</td>
<td>215</td>
</tr>
<tr>
<td>134</td>
<td>Viewpoint Definition View showing the elements that appear on the Traceability Identification View (TIV)</td>
<td>216</td>
</tr>
<tr>
<td>135</td>
<td>An Example of a Traceability Identification View for MBRE</td>
<td>218</td>
</tr>
<tr>
<td>136</td>
<td>Viewpoint Context View showing Traceability Viewpoint aims</td>
<td>219</td>
</tr>
<tr>
<td>137</td>
<td>Viewpoint Definition View showing the elements that appear on the Traceability View (TV)</td>
<td>220</td>
</tr>
<tr>
<td>138</td>
<td>An example of a Traceability View</td>
<td>221</td>
</tr>
<tr>
<td>139</td>
<td>Viewpoint Context View showing Impact Viewpoint aims</td>
<td>223</td>
</tr>
<tr>
<td>140</td>
<td>Viewpoint Definition View showing the elements that appear on the Impact View (IV)</td>
<td>224</td>
</tr>
<tr>
<td>141</td>
<td>An example of an Impact View</td>
<td>225</td>
</tr>
<tr>
<td>142</td>
<td>Architectural Framework Context View showing Life Cycle Pattern aims</td>
<td>229</td>
</tr>
<tr>
<td>143</td>
<td>Ontology Definition View showing Life Cycle concepts</td>
<td>230</td>
</tr>
<tr>
<td>144</td>
<td>Viewpoint Relationship View showing Life Cycle Viewpoints</td>
<td>231</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>145</td>
<td>Rules Definition View showing Life Cycle rules</td>
<td>232</td>
</tr>
<tr>
<td>146</td>
<td>Viewpoint Context View showing Life Cycle Viewpoint aims</td>
<td>232</td>
</tr>
<tr>
<td>147</td>
<td>Viewpoint Definition View showing the elements that appear on the Life Cycle Viewpoint (LCV)</td>
<td>233</td>
</tr>
<tr>
<td>148</td>
<td>An example Life Cycle View</td>
<td>233</td>
</tr>
<tr>
<td>149</td>
<td>Viewpoint Context View showing Life Cycle Model Viewpoint aims</td>
<td>234</td>
</tr>
<tr>
<td>150</td>
<td>Viewpoint Definition View showing the elements that appear on the Life Cycle Model Viewpoint (LCMV)</td>
<td>235</td>
</tr>
<tr>
<td>151</td>
<td>An example of a Life Cycle Model View</td>
<td>236</td>
</tr>
<tr>
<td>152</td>
<td>Viewpoint Context View showing Interaction Identification Viewpoint aims</td>
<td>237</td>
</tr>
<tr>
<td>153</td>
<td>Viewpoint Definition View showing the elements that appear on the Interaction Identification Viewpoint (IIV)</td>
<td>237</td>
</tr>
<tr>
<td>154</td>
<td>An example of an Interaction Identification View</td>
<td>238</td>
</tr>
<tr>
<td>155</td>
<td>Viewpoint Context View showing Interaction Behaviour Viewpoint aims</td>
<td>239</td>
</tr>
<tr>
<td>156</td>
<td>Viewpoint Definition View showing the elements that appear on the Interaction Behaviour Viewpoint (IBV)</td>
<td>239</td>
</tr>
<tr>
<td>157</td>
<td>An example of an Interaction Behaviour View</td>
<td>240</td>
</tr>
<tr>
<td>158</td>
<td>An example of an Interaction Behaviour View showing full Life Cycle Interactions</td>
<td>241</td>
</tr>
<tr>
<td>159</td>
<td>Architectural Framework Context View showing Epoch aims</td>
<td>242</td>
</tr>
<tr>
<td>160</td>
<td>Ontology Definition View showing Epoch concepts</td>
<td>243</td>
</tr>
<tr>
<td>161</td>
<td>Viewpoint Relationships View showing Epoch Viewpoints</td>
<td>245</td>
</tr>
<tr>
<td>162</td>
<td>Rules Definition View showing Epoch Pattern rules</td>
<td>246</td>
</tr>
<tr>
<td>163</td>
<td>Viewpoint Context View showing Epoch Definition Viewpoint aims</td>
<td>247</td>
</tr>
<tr>
<td>164</td>
<td>Viewpoint Definition View showing the elements that appear on the Epoch Definition Viewpoint (EDV)</td>
<td>248</td>
</tr>
<tr>
<td>165</td>
<td>An example of an Epoch Definition View showing two Epochs</td>
<td>248</td>
</tr>
<tr>
<td>166</td>
<td>Viewpoint Context View showing Applicable Viewset Viewpoint aims</td>
<td>249</td>
</tr>
<tr>
<td>167</td>
<td>Viewpoint Definition View showing the elements that appear on the Applicable Viewset Viewpoint (AVV)</td>
<td>250</td>
</tr>
<tr>
<td>168</td>
<td>An example of an Applicable Viewset View showing Service-Based Interfaces</td>
<td>251</td>
</tr>
<tr>
<td>169</td>
<td>Viewpoint Context View showing Metric Definition Viewpoint aims</td>
<td>252</td>
</tr>
<tr>
<td>170</td>
<td>Viewpoint Definition View showing the elements that appear on the Metric Definition Viewpoint (MDV)</td>
<td>253</td>
</tr>
<tr>
<td>171</td>
<td>An example of a Metric Definition View</td>
<td>254</td>
</tr>
<tr>
<td>172</td>
<td>Viewpoint Context View showing Metric Usage Viewpoint aims</td>
<td>256</td>
</tr>
<tr>
<td>173</td>
<td>Viewpoint Definition View showing the elements that appear on the Metric Usages Viewpoint (MUV)</td>
<td>256</td>
</tr>
<tr>
<td>174</td>
<td>An example of a Metric Usage View</td>
<td>257</td>
</tr>
<tr>
<td>175</td>
<td>Architectural Framework Context View showing Contract Pattern aims</td>
<td>259</td>
</tr>
<tr>
<td>176</td>
<td>Ontology Definition View showing Contract Pattern concepts</td>
<td>260</td>
</tr>
</tbody>
</table>
Figure 177 - Viewpoint Relationship View showing Contract Pattern viewpoints .................................................................261
Figure 178 - Selection of rules governing the use of the Contract Pattern ...........262
Figure 179 - Context of the Contractual SoS Definition Viewpoint .........................263
Figure 180 - Viewpoint Definition View showing the elements that appear on the Contractual SoS Definition Viewpoint (CSDV) ..................................................................................263
Figure 181 - An example of a Contractual SoS Definition View showing the contracts in a AV Contractual SoS ..................................................................................................................264
Figure 182 - Context of the Contract Conformance Viewpoint ................................265
Figure 183 - Viewpoint Definition View showing the elements that appear on the Contract Conformance View (CCV) ........................................................................................................265
Figure 184 - Contract Conformance View example showing the conformance relationship between the elements in an AV Case Study ..............................................................266
Figure 185 - Context of the Contract Connections Viewpoint ..................................267
Figure 186 - Viewpoint Definition View showing the elements that appear on the Contract Connections Viewpoint (CConnV) .................................................................267
Figure 187 - An example of a Contract Connections View showing the connections between two AV Device and one Transport contract .......................268
Figure 188 - Context of the Contract Definition Viewpoint ....................................269
Figure 189 - Viewpoint Definition View showing the elements that appear on the Contract Definition View (CDV) .............................................................................................269
Figure 190 - An example Contract Definition View ................................................270
Figure 191 - Contract Definition View defined using CML ...................................271
Figure 192 - Context of the Contract Protocol Definition Viewpoint .....................272
Figure 193 - Viewpoint Definition View showing the elements that appear on the Contract Protocol Definition View (CPV) ..............................................................272
Figure 194 - Example of a Contract Protocol Definition View ................................273
Figure 195 - Example Contract Protocol Definition View defined using CML ......274
Figure 196 - System block containing both TE and SUT ....................................289
Figure 197 - Internal block diagram specifying the interface between TE and SUT .................................................................................................................................289
Figure 198 - Functional decomposition of SUT into indication function control and output control .................................................................................................290
Figure 199 - Hierarchic state machine FLASH_CTRL ........................................292
Figure 200 - Hierarchic state machine OUTPUT_CONTROL .............................293
Figure 201 - Example of a traceability view from requirements to test cases ....296
Figure 202. Block definition diagram of abstract model .......................................303
Figure 203. State machine diagram of the abstract model ....................................303
Figure 204. Block definition diagram of concrete model ......................................303
Figure 205. Internal block diagram of concrete model ........................................304
Figure 206. State machine diagram of block Sec of the concrete model ............305
Figure 207. State machine diagram of block Min of the concrete model ............305
Figure 208. The Circus refinement strategy ..........................................................308
Tables

Table 1 - Comparison of terms in defence-based architectural frameworks .......... 31
Table 2: SoS Examples and respective patterns ............................................. 85
Table 3 - Document structure for an architectural pattern................................ 90
Table 4 - An example of a text-based Relationship Identification View for MBRE .......................................................................................................................... 215
Table 5 - An example of a text-based Traceability Identification View for MBRE ....................................................................................................................... 217
Table 6 – An example of a Traceability View represented as a matrix ............ 222
Table 7 – An example of an Impact View represented using text ................... 226
Table 8- Requirements list for the turn indication function .............................. 289
Table 9 - Traceability matrix relating requirements to model elements .......... 295
1. Introduction

This document contains the results of a study into modelling patterns for System of Systems (SoS) architectures. The following subsections establish the scope of the document and the context in which it has been produced.

1.1. Scope

This document presents a discussion of architectural principles for SoS, a classification of different SoS types, a collection of architectural styles and patterns for SoS and a discussion of a refinement calculus for SysML that aims to support the formal analysis of SysML models. The principal aim of the document is to present a collection of patterns that can be used in the architectural modelling of a SoS, both in terms of architectural style and also in terms of typical systems engineering activities such as the definition of interfaces or the establishing of traceability.

1.2. Context

Architectural design is seen as an essential part of systems engineering and is one of the key technical processes in ISO15288:2008 “Systems and software engineering – System life cycle processes” [ISO15288:2008], a widely-adopted systems engineering standard.

An architectural design “provides a level of abstraction which allows designers to reason about system behavior” [Stevens et al 1998], identifying and exploring “one or more implementation strategies at a level of detail consistent with the system's technical and commercial requirements and risks” [ISO15288:2008].

An architectural design must cover three key areas: [Stevens et al 1998]

- System structure, defining the major components of the system, their organisation and structure.
- System behaviour, defining the “dynamic response of the system to events, providing a basis for reasoning about the system.” (our italics)
- System layout, defining the physical layout and packaging of the system.

TWO key enablers for the production of an architectural design are architectural frameworks and modelling patterns.

An architectural framework specifies a number of views of an architecture that must be produced when creating the architectural design. They are an aid to the architectural design process as they force the engineer to consider different views on the architecture. However, architectural frameworks are usually created for a specific purpose (such as military acquisition or enterprise architecture) and as such the views that they define are those that are deemed necessary to meet the requirements of that framework. This means that not all
the views in a given framework may be relevant or that a framework may be missing needed views.

A pattern is a proven best-practice solution to a known, recurring problem within a given context that has been used many times and which can be adapted to suit the needs of a particular situation. The term *modelling pattern* simply refers to a pattern that can be applied to modelling aspects of a system, such as its architecture or its interfaces, or that can be applied to the systems engineering process, such as patterns for requirements engineering.

Architectural frameworks were surveyed in COMPASS deliverable D22.1 [D22.1 2012]. This document focuses on underlying architectural principles and patterns, taken from the domains of systems and software engineering, and discusses how such principles and patterns apply to, or may be extended to cover, SoSs. The identification and documenting of patterns allows architects to learn from previous experiences, capturing experiences that have proven successful in the past [Kalawsky et al 2013], and improves reuse of quality designs.

However, principles and patterns on their own are not sufficient. While they can help in the selection of appropriate patterns for particular types of SoS and can guide in the definition of system structure and behaviour, they do not directly address the reasoning about the system, seen by [Stevens et al 1998] as a key aspect of the system behaviour part of an architectural design. Also, architectures are rarely fixed; systems evolve over time and their architectures must also change. This document, therefore, also considers *transformation of* architectural designs both as a response to changing requirements and system evolution and also to lower-level designs. In addition, the document discusses support for *reasoning* about architectural patterns and the architectures built using them.

### 1.3. Document Structure

Following this Introduction, in Section 2 we present a discussion of architectural principles for SoS. The section begins with a discussion on definitions relating to architecture and architectural frameworks then considers architectural techniques for both systems and SoSs from a variety of information sources. These definitions and techniques are then abstracted into a coherent and consistent set for use in SoS engineering. Section 3 looks at the types of SoS from a range of application domains including those covered in the COMPASS case studies and includes taxonomies for different types of SoS and architectural models. In Section 4 a collection of patterns is presented. Two different types of patterns are considered: *architectural* patterns that describe specific system architectures (such as a service oriented architecture) and *enabling* patterns whose use enables a number of systems engineering applications (such as an interface definition pattern). Section 5 presents the summary of this report and Section 6 completes the main body of the document with the references used. Finally, Appendix A relates the automated model-based testing work that is being done on COMPASS with the Test Pattern discussed in Section 0 and
Appendix B introduces the SysML semantics formalised in D22.4, notions of refinement suitable for SysML based on the formal semantics, and an initial catalogue of SysML refinement laws.
2. Discussion of Architectural Principles for SoS

This section presents a discussion of architectural principles for SoS. It consists of four main subsections:

- Section 2.1 considers a number of sources to investigate fundamental definitions of architecture and architectural framework.
- Section 2.2 considers a number of sources to investigate architectural techniques as they apply to systems.
- Section 2.3 considers a number of sources to investigate architectural techniques as they apply to systems of systems.
- Section 2.4 then abstracts the concepts and techniques covered in Sections 2.1 to 2.3, comparing, contrasting and combining into a coherent set of definitions and principles.

Following these four main subsections, Section 2 concludes with a summary.

2.1. Fundamental Definitions

The concept of an architecture is fundamental to any systems engineering undertaking. There is much confusion, however, due to the plethora of very similar terms involving the word ‘architecture’. The use of architecture has also changed significantly over the years as the definitions that apply to the world of software engineering have a significantly-simpler scope than the world of systems engineering. A related concept is that of the architectural framework. This section explores these concepts by investigating definitions of both from a number of sources. Common concepts are abstracted and used to give a definition of these terms as relevant to COMPASS and SoS engineering.

2.1.1. Concepts from Systems and Software Engineering Literature

Many definitions of the term “architecture” can be found in systems and software engineering literature, including the following:

- ‘The structure of levels and/or branches that partition a system into its constituent parts or components’ [Stevens et al 1998]
- ‘modules and how they are interconnected’ [Schach 1997]
- ‘relationship among major components of the program’ [Pressman 2000]
- ‘a hierarchy of components according to a partitioning method’ [Sanders&Curran 1994]
- ‘The set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour, as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems and the architectural style that guides this organization’ [Booch et al 2005]

In summary, an architecture defines the major elements of a system, identifies the relationships and interactions between the elements and takes into account
An architecture involves both a definition of structure and behaviour. Importantly, architectures are not static but must evolve over time to reflect the change in a system as it evolves to meet changes to its requirements.

2.1.2. Concepts as described by the International Standards Organisation (ISO)

There are two main ISO standards that are relevant to the definition of architecture:

- ISO 42010 – Systems and software engineering - Architecture description [ISO42010:2007]. This standard is more recent than ISO 15288 and is concerned solely with architectures. The terminology provided in this standard, therefore, is very well defined and has far more detail than in ISO 15288.

In ISO 15288, the definition of an architecture is given as:

‘fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.’

ISO/IEC 42010 defines a number of terms:

- architecting: process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout a system’s life cycle
- architecture: fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution
- architecture description (abbreviation 'AD'): work product used to express an architecture
- architecture description language (abbreviation 'ADL'): any form of expression for use in architecture descriptions
- architecture framework: conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders
- architecture viewpoint: work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns
- architecture view: work product expressing the architecture of a system from the perspective of specific system concerns
- concern: interest in a system relevant to one or more of its stakeholders. A concern pertains to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences.
• stakeholder: individual, team, organization, or classes thereof, having an interest in a system

The fundamental definition of the term ‘architecture’ is almost identical between the two standards, but ISO 42010 provides far more definitions that ISO 15288.

Figure 2 shows a summary of the architecture-related terms for ISO 42010. As the definition of ‘architecture’ in ISO 15288 is almost identical, then this diagram can be seen to represent both standards.

2.1.3. Concepts as described by the International Council on Systems Engineering (INCOSE)

INCOSE have produced the ‘Systems Engineering Handbook’ [INCOSE 2011] which is a systems engineering body of knowledge and commentary on ISO 15288.

The INCOSE handbook does not provide an explicit definition of what an architecture is, yet does refer to several different types of architecture, such as: system architecture, logical architecture, functional architecture, enterprise architecture and also introduces the term architectural (architecture) framework:

‘there may be several required operational views of the system driven by architectural frameworks’
2.1.4. Concepts as described by the architectural framework community

The previous three sections have explored the concept of architecture. When architectures are used for a specific application domain, industry, or stakeholder group, it is quite common for an architectural (or architecture) framework to be defined. An architectural framework is not, in itself, an architecture, but defines a set of views that are required to describe an architecture. Note that the terms architectural framework and architecture framework are both used to mean the same thing and, often, both used interchangeably in the same source material.

A number of architectural frameworks are considered here at a very high level. For a more in-depth description see [Holt&Perry2010].

2.1.4.1. The Zachman Framework

The Zachman Framework is a framework for Enterprise Architecture. The Zachman Framework is one of the oldest and most mature frameworks and is certainly one of the most widely-used in industry today.

The framework itself takes the form of a simple matrix, comprising rows and columns with intersecting cells that describe aspects of an entity. Usually there are 36 cells as the matrix has six rows and six columns. [Zachman2008].

Each row represents a particular perspective on an enterprise architecture (such as the Executive Perspective or the Engineer Perspective). Each column asks a particular question of the enterprise architecture (What? How? Where? etc). The cells at the intersection of each row and column represent the part of the model that answers the question (represented by the column) for that perspective (represented by the row). For example, answering the How? question for the Engineer Perspective leads to the production of the Process Specification model. Answering the Who? question for the Executive Perspective leads to the production of the Responsibility Identification model.

2.1.4.2. Defence-based architecture frameworks

The defence industry has a number of architectural frameworks that originate in and are used by particular countries. These include:

- MODAF – Ministry of Defence Architecture Framework [MODAF2010] which originated and is used in the UK.
- DoDAF – Department of Defense Architecture Framework [DoDAF2007] which originated and is used in the USA.
- NAF – NATO Architecture Framework [NAF2007], which is used in NATO countries.
- DNDAF – DND/CF Architecture Framework, which originated and is used in Canada [DNDAF 2012]
Many other such frameworks exist, with many countries having their own specific framework. Most of them are closely related and, therefore, have very similar constructs and concepts. Here, a single framework, the UK Ministry of Defence Architectural Framework (MODAF), will be considered. MODAF defines an architectural framework as:

‘An Architectural Framework (AF) is a specification of how to organise and present architectural models. ... an AF defines a standard set of model categories (called 'Views') which each have a specific purpose. These views are often categorised by the domain they cover – e.g. operational / business, technical, etc. – which are known in MODAF as Viewpoints’

In terms of the application of MODAF, its scope includes:

‘MODAF provides a rigorous method for understanding, analysing, and specifying: Capabilities, Systems, Systems of Systems (SoS), Organisational Structures and Business Processes.’

The MODAF specification identifies seven viewpoints that are required to make up the full architectural framework. Note that this is a different definition of the term 'Viewpoint' from the one used in ISO 42010.

The MODAF is made up of seven Viewpoints each of which is made up of one or more Views. These Viewpoints are simply collections of Views and serve to group Views that are used for similar purposes. The Viewpoints are:

- **All Views** viewpoint that describe information that applies to all viewpoints.
- **Strategic** viewpoint that describes required system capabilities.
- **Operational** viewpoint that describes operational concepts.
- **System** viewpoint that describes the actual systems to be delivered.
- **Service-oriented** viewpoint that describes the actual services that form part of the delivered systems.

![Figure 3 - Summary of the MODAF architecture-related concepts](image-url)
The various defence-based frameworks are related to each other and so have a number of similarities. The following table provides a high-level mapping between the terms that are used in several frameworks.

<table>
<thead>
<tr>
<th>MODAF</th>
<th>DoDAF</th>
<th>NAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewpoint</td>
<td>View</td>
<td>View</td>
</tr>
<tr>
<td>View</td>
<td>Product</td>
<td>Subview</td>
</tr>
<tr>
<td>All Viewpoint</td>
<td>All View</td>
<td>NATO All View</td>
</tr>
<tr>
<td>Acquisition Viewpoint</td>
<td>-</td>
<td>NATO Programme View</td>
</tr>
<tr>
<td>Strategic Viewpoint</td>
<td>-</td>
<td>NATO Capability View</td>
</tr>
<tr>
<td>Operational Viewpoint</td>
<td>Operational View</td>
<td>NATO Operational View</td>
</tr>
<tr>
<td>Systems Viewpoint</td>
<td>Systems and Services View</td>
<td>NATO Systems View</td>
</tr>
<tr>
<td>Service-oriented Viewpoint</td>
<td>Systems and Services View</td>
<td>NATO Service-oriented View</td>
</tr>
<tr>
<td>Technical Viewpoint</td>
<td>Technical Standards View</td>
<td>NATO Technical view</td>
</tr>
</tbody>
</table>

Table 1 - Comparison of terms in defence-based architectural frameworks

Table 1 is not intended to be an exhaustive comparison but is intended to the main similarities between the various defence-based architectural frameworks. For a full discussion on the similarities and differences between these frameworks, including models of the different structures, see [Holt&Perry2010].

2.1.4.3. Non-defence architecture frameworks

Alongside the defence-based architectural frameworks, there are a number of non-defence architecture frameworks (note the difference in terms here, where the term ‘architectural framework’ was used in defence, the term ‘architecture framework’ is used here; both mean the same thing).

Perhaps the most widely-known is The Open Group Architecture Framework (TOGAF) [TOGAF 2012] which is not actually an architecture framework but, rather, a set of phases and associated processes in the form of an architecture development method (ADM) that enables an enterprise architecture to be created for an organisation. TOGAF does not define any particular views but focuses on how to manage the development and delivery of the architecture.

Another example of a non-defence architecture framework is TRAK that was originally commissioned by London Underground Limited in the UK but has since been adopted by a number of organisations [TRAK 2012]. TRAK cites both ISO 15288 and ISO 42010 as major references and it has a strong systems engineering flavour. Figure 4 shows an overview of TRAK.
The diagram shows that TRAK is made up of five Perspectives that group one or more related Views, and 21 Viewpoints. Each View is defined according to one or more Viewpoint.

2.2. Existing Systems-level Architectural Techniques

This section considers a number of sources to investigate architectural techniques as they apply to systems. Five key information sources are considered:

- Information on the Object Management Group’s model-driven architecture.
- The book ‘Architecture and Principles of Systems Engineering’ by Dickerson and Mavris, a text widely used in industry.
- The international standard ISO 42010 ‘Systems and software engineering – Architecture description’.
- Relevant academic sources.

The key principles and techniques regarding architectural techniques as they are applied to systems are identified from each information source. These principles and techniques, along with the fundamental definitions identified in Section 2.1 and the SoS techniques identified in the following section will be compared, contrasted and combined into a coherent set in Section 2.4.

2.2.1. Model-Driven Architecture

The Model Driven Architecture (MDA) is “a framework for software development defined by the Object Management Group (OMG)” [Kleppe et al 2003]. It aims to address issues of productivity, portability, interoperability and maintenance by clearly separating the concepts of the description of a system from the description of how that system is to be implemented and by automating the transformation between these descriptions. For the latest MDA specifications see [MDA 2013].

An overview, based on [Kleppe et al 2003] is shown in Figure 5:
Key to the MDA is the concept of a ‘Model’, which is a description of a ‘System’, and which is written in a ‘Language’ (which could, for example, be a modelling language like SysML, a formal language like CML or a programming language like Java).

MDA also defines the concept of a ‘Transformation Definition’, made up of one or more ‘Transformation Rule’. Such a ‘Transformation Definition’ describes how a source ‘Language’ can be transformed into a target ‘Language’. Such transformations are carried out by a ‘Transformation Tool’ that executes one or more ‘Transformation Definition’.

This concept of transformation is key to MDA and is applied to the three main types of Model defined by the MDA:

- The ‘Platform Independent Model (PIM)’, which describes a system without any consideration of the final implementation platform.
- The ‘Platform Specific Model (PSM)’, which describes a system assuming a particular implementation platform.
- The ‘Code’, which represents the source code that implements the software components of the ‘System’.

The idea behind MDA is that the analysis work is done at the PIM-level, allowing modelling to be carried out using concepts relevant to the domain of the system and then *automatically transformed* into one or more PSMs and from them into code, by means of transformation tools execution transformation definitions.
For example, the PIM for an on-line shopping system would contain concepts such as Customers, Orders and Items. If it is decided that the system is to be implemented as a three-tier architecture, then the PIM would be transformed into three PSMs: for example, one for a database tier implemented using relational databases, one for a middle tier using an Enterprise Java Beans (EJB) platform and one for a front-end using a Web-based platform. Each of these PSMs would then also be transformed into the relevant code, such as SQL for the database, EJB source code for the middle tier and JSP source code for the front-end.

Although the MDA is aimed specifically at the development of software-systems, it contains a number of concepts that are applicable to more general systems engineering:

- **Models** are central to the successful development of a system.
- A number of *models* may be required to fully define a system.
- Models should be produced at a particular level of abstraction. In the MDA, the PIM is at the highest level of abstraction, the PSMs at an intermediate level and the code at the lowest level.
- System development involves the refinement of models from high to low-levels of abstraction.
- (Semi-)automatic transformation of one model to another is a powerful technique for creating consistent models represented using different languages that allow different analysis techniques to be applied.

Although MDA posits complete model-to-model transformation, the same powerful ideas can be applied to parts of a model. For example, a system (or system of systems) could have its interfaces modelled using SysML and the *Interface Definition* enabling pattern (see Section 4.4.1). This gives a good, semi-formal graphical and textual definition of the interfaces. Such a model readily supports systems-engineering concepts such as *traceability* and is in a representation widely used within the systems engineering community. However, such a model of the interfaces is not readily amenable to any kind of *formal* analysis. Transforming just the interface part of the model into a model written in a formal language (CML, VDM etc.) gives a representation of the interfaces that *is* amenable to formal analysis. With bidirectional transformation definitions, any changes subsequently made in the formal model can then be automatically pushed back into the SysML model. Indeed, this concept of working with two different representations of (aspects of) a system is at the heart of the COMPASS project.

### 2.2.2. Abstraction From Dickerson & Mavris “Architecture And Principles Of Systems Engineering”

The book “Architecture and Principles of Systems Engineering” by Dickerson and Mavris [Dickerson & Mavris 2009] provides a wide-ranging discussion on the
state of both systems engineering and SoS engineering. It provides definitions of terms, discusses relevant standards and provides an overview of relevant techniques such as the Unified Modelling Language (UML), the Systems Engineering Modelling Language (SysML) and architectural frameworks including DoDAF and MODAF. Although it doesn’t directly define (or even outline) a process for producing an architecture, a number of relevant points can be abstracted. The quotes in this section are taken from [Dickerson & Mavris 2009] and also give the page numbers from that book.

At a general systems engineering level:

- Systems engineering is seen as a key enabler for the delivery of capability: “... the ability to execute a specified course of action that is defined by a user and is expressed in non-equipment-based operational terms.” {p42; definition from UK MOD}. Although this definition originates in the defence-domain it is no less relevant to the wider systems engineering community. All systems are developed because their procurers have a capability that they need (or perceive that they need) to be able to deliver.

- Model-Driven Architecture (MDA - see previous section) and Model-Based Systems Engineering (MBSE) “will play an important role in determining how the practice of architecture and systems engineering evolves over the next several years.” {p3}

With respect to architectures:

- The authors provide a number of definitions of the term architecture. The one that is used most often throughout the book states that “The architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.” {p39; definition taken from Miller & Mukerji’s 2003 OMG MDA Guide}.

- Modelling is seen as essential for the production of architectures; the “specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors is realized through the use of models.” {p403}. UML, SysML, mathematical models, propositional calculus etc. can all be used when creating architectures {Chapters 4 to 9}

- Three different types of architecture are discussed, originating from the US Department of Defence (DoD) systems engineering process, which utilises these architectures “to describe and define different aspects of the system under development”. The three types described are:
  - Functional architecture – a structured description of the system’s functionality. It “embodies the structure of allocated performance and functional requirements”. {p277, 279}
  - Physical architecture – the structural breakdown of the physical system into “various levels of subsystems, components and parts”. The architecture thus embodied must perform “all the functions
with the prescribed required performance as established in the functional architecture.” Thus, when creating a physical architecture, it is essential “that all components contribute to meeting one or more functional requirements.” {p277, 280, 281} Thus, it must be possible to allocate all functional elements from the functional architecture to components of the physical architecture.

- System architecture – which “identifies all products and processes required to support the system across its entire life cycle”. The system architecture is “a description of the entire system based on the physical architecture and the definition of supporting products and services across all phases of the life cycle.” {p277, 282}

- The levels of decomposition of the functional and physical architectures that are mapped on to another should be the same. This is not a one-off process, but done in a series of “cycles” that decompose the definition of the system (through its architectures) into ever more detailed representations, with each level of decomposition verified against the previous. {p303}

- Physical and functional architectures are often used to support trade-off analysis in order to establish which of a potential number of candidate architectures best meets the customer requirements at a particular level of decomposition. When the candidate solution has been verified against requirements, the “result is a system-level specification and the system is thus defined by its functional and physical architectures.” {p311}. This is part of the larger concept of architecture-based assessment, in which the architecture of a system forms the base artefact against which the proposed solution meets its requirements. {p441}

- The use of executable architectures allow simulations of the system to be made and problems with the architecture to be quickly found and corrected. {p414}. However, there is as yet no widely accepted notation for defining executable architectures. {p403}

- Architectural reuse is seen as important, as it “allows projects to quickly identify conceptually similar existing architectures and quickly interpret them to the chosen application.”{p415}

With respect to architectural frameworks:

- Architectural frameworks (AFs) are seen as an important tool that aid in the production of architectures. They “…are a basic underlying relational structure for forming or constructing an architecture.” {p155}

- While many of the most well-known AFs are used in the defence domain (DoDAF in the US, MODAF in the UK, NAF across NATO), AFs are considered to be essential in all domains; for example, the US has the Federal Enterprise Architecture Framework (FEAF) that is used by federal agencies outside the DoD. The creation and widespread adoption of AFs in the US is a direct response to the 1996 Clinger-
Cohen Act which “precipitated the use and generation of architecture frameworks by many agencies of the government.” {p189}

2.2.3. ISO42010

ISO 42010 ‘Systems and software engineering – Architecture description’ does not explicitly define processes for the production of an architecture. Indeed, the latest version of the standard states ‘This International Standard does not prescribe the process or method used to produce architecture descriptions. This International Standard does not assume or prescribe specific architecting methods, models, notations or techniques used to produce architecture descriptions.’ [ISO42010:2011]. Rather, it concentrates on what it refers to as architecture descriptions; what they are, what they can be used for and how they are structured.

In the terminology used in ISO 42010, every system exhibits an architecture which is considered to be abstract, consisting of concepts and properties relating to the system. Such an abstract architecture is realised by an architecture description (the concrete work product) that is made up of a number of architecture views that address the concerns of stakeholders. Each architecture view is made up of a number of architecture models and conforms to an architecture viewpoint that defines the allowed content of the view.

ISO 42010 emphasises three key points about architectures and architecting:

1. Although the standard focuses on systems and software, architecture descriptions can be produced for use outside these domains: ‘nothing herein precludes its use for architecture descriptions of entities of interest outside of those domains (for example, natural systems and conceptual systems).’
2. Architecting takes place throughout the life cycle of a system: ‘Architecting is performed throughout the system life cycle, not simply within one stage of the life cycle. Therefore, a system’s architecture potentially influences processes throughout the system’s life cycle.’
3. Architecture descriptions have a wide range of uses beyond the usual use as a basis for system design and evaluation of alternative implementations, including: as design documentation, as input to simulation and analysis tools, as a communication tool between parties involved in the system (both on the supply and acquisition side), as an aid to support and infrastructure planning and implementation, as a basis of system review and evaluation throughout its life cycle.

Although ISO 42010 does not explicitly define a process for developing an architectural description, the following tasks can be abstracted from the standard:

- **Identify stakeholders** – identify stakeholders that have concerns to be addressed by the architecture.
- **Elicit concerns** – elicit the concerns from the stakeholders.
• **Identify the relevant architectural viewpoints** – identify the architectural viewpoints that will be used in the production of the architecture description. These can be viewpoints that already exist in an architecture framework, or viewpoints that will be created as part of the architecture description. Each concern must be addressed by at least one viewpoint.

• **Record rationale** – the rationale for the inclusion of each architectural viewpoint must be recorded.

• **Define architectural viewpoints** – if suitable architectural viewpoints do not exist, then they will be created as part of the architecting work. Any such architectural viewpoints can be fully defined as part of the architectural description, documented as an architectural framework or created as individual viewpoints and added to a reusable viewpoint library. ISO 42010 gives guidance on how an architectural viewpoint should be documented. Definition of architecture viewpoints is considered in COMPASS deliverable D21.2 'Initial Report on Guidelines Architectural Level SoS Modelling'.

• **Produce architectural views** – one architecture view will be produced for each specified architectural viewpoint. Each view is represented by one or more architectural models (and a model can be shared across multiple views). The use of multiple models within a view allows different aspects of the view to be captured in each model; the totality of models for a given view must ‘address all of the concerns framed by its governing viewpoint and cover the whole system from that viewpoint’.

• **Establish consistency** – establish and define consistency rules (termed correspondence rules in ISO 42010) between the various views produced and the information contained in them. For example, in the Interface Definition pattern defined in Section 4.4.1 below, Rule ID2 (along with the other rules defined for this pattern) is an example of such a correspondence rule. Conformance to and variation from these correspondence rules should be documented as part of the architecture description.

The outputs of these tasks form the various parts of the architectural description.

### 2.2.4. ISO15288 Architectural Design Process

ISO 15288 defines an Architectural Design Process, the purpose of which is ‘is to synthesize a solution that satisfies system requirements’ [ISO15288:2008].

The process ‘encapsulates and defines areas of solution expressed as a set of separate problems of manageable, conceptual and, ultimately, realizable proportions. It identifies and explores one or more implementation strategies at a level of detail consistent with the system’s technical and commercial requirements and risks. From this, an architectural design solution is defined in terms of the requirements for the set of system elements from which the system is configured. The specified design requirements resulting from this process are the basis for verifying the realized system and for devising an assembly and verification strategy.’
The process is summarised in Figure 6.

As can be seen from Figure 6, the Architectural Design Process is made up of three activities, each broken down into a number of tasks. The process also has a number of defined outcomes.

The three activities are:

- **Define the architecture** – appropriate logical architectural designs are created, including the definition of any derived requirements, and traced back to defined system requirements. System functions are allocated to elements of the architecture. Interfaces between system elements and cross-boundary interfaces with external systems are defined.

- **Analyze and evaluate the architecture** – the architecture is analysed in order to establish design criteria for each element. Design criteria include ‘physical, performance, behavioural, durability and sustainable service characteristics’. Those system requirements that can be allocated to human operators are determined and a determination is made as to whether off-the-shelf components can be used. Alternative design solutions are modelled and evaluated.

- **Document and maintain the architecture** – the selected architecture is baselined, giving a basis against which the system solution can be developed. The architecture baseline is specified in terms of ‘its functions, performance, behaviour, interfaces and unavoidable implementation constraints.’ The architectural design information is recorded, capturing
the structural and functional aspects, interfaces, design decision and conclusions, all traced to the requirement baseline. Traceability between system design and system requirements is maintained.

The outcomes of the process are:
- A baselined architectural design
- Specified system elements, traced to system requirements, that can be implemented
- Interface requirements captured and incorporated into the architectural design
- Traceability of architectural design to system requirements captured
- Basis for system element verification established
- Basis for integration of system elements established

ISO 15288 directs readers to ISO 42010 for guidance on the representation of architectures.

2.2.5. Academic search

Architectural techniques attracting interest in the academic community include:
- Architectural styles
- Architecture and agile development
- Component-based software engineering and distributed systems
- Architectural Description Languages (ADLs)

Architectural styles
An architectural style “defines a family of such systems in terms of a pattern of structural organization” [Garlan & Shaw 1994]. Describing systems in terms of architectural style facilitates reasoning and understanding about a system's design, by providing a set of high-level structural properties, a vocabulary to describe them, and some assumptions and constraints on how they should be employed [Garlan 2000]. Well-known styles include:
- pipe-and-filter style [Garlan & Shaw 1994; Monroe et al 1997], in which components are represented as “filters” connected serially by “pipes” which provide input and output channels for data
- “blackboard” style [Garlan & Shaw 1994; Monroe et al 1997], in which components are represented as modules clustered around a central data repository - or “blackboard” - each interacting with the blackboard directly
- object-oriented [Garlan & Shaw 1994], in which data representations and primitive operations on them are encapsulated in modules or abstract data types
- event-based or publish-subscribe architectural styles, which may involve explicit or implicit invocations [Garlan & Shaw 1994]
- layered systems [Garlan & Shaw 1994], such as the well-known protocol stacks employed by network standards, in which functions are gathered into “layers” that provide services to the layer logically positioned above, and consume services to the layer logically positioned below.
Service-oriented architecture (SOA), which is an architectural style where components of the system may act as service users in some instances and service providers in others. The services are expected to be self-contained and capable of being independently deployed; are distributed and typically accessed over a network; have published interfaces, usually with implementation details hidden; are interoperable and substitutable; are discoverable; and is dynamically bound (i.e., the particular service implementation is discoverable at runtime).

The benefits of applying a recognised architectural style to a system include:

- The system design which implements an architectural style benefits from “lessons learned” on previous systems with similar architectural properties. Potential or common pitfalls can be avoided. The styles are “design guides” [Shaw 2001].
- The design is easier to understand, as it will be applying a well-known paradigm that can be recognised by personnel not necessarily well-versed in the problem domain [Monroe et al 1997].
- There are increased chances that code and/or designs will be reused, since there will be other systems employing the same architectural structure which can be examined for potentially reusable modules [Monroe et al 1997].
- It will be easier to build on a standard architectural framework – thus increasing interoperability - if a common architectural style is selected [Monroe et al 1997].
- Employing a recognised architectural style allows for specialised reasoning and/or analysis to be applied [Monroe et al 1997].

Whilst the well-known styles listed above have been described, Garlan & Shaw [Garlan & Shaw 1994] point out that many systems can be viewed as implementing several styles, depending on the scope and granularity of the current view.

As already described, an architectural style provides a vocabulary for describing a family of systems that have some structural properties in common. This is distinct from design patterns, although the fields of architectural styles, design and patterns overlap significantly. In particular, design patterns (such as those introduced by Gamma et al [Gamma et al 1995]) and architectural styles can be viewed as distinct entities. An architectural view of a system provides a high-level view that enables the analysis of “emergent system wide properties such as freedom from deadlock (provided that the system contains no cycles), throughput rates, and potential system bottlenecks” [Monroe et al 1997]. In contrast, design patterns tend to be concerned much more with lower-level questions of how subsets of components interact. “Styles generally provide guidance and analysis for building a broad class of architectures in a specific domain, whereas patterns focus on solving smaller, more specific problems within a given style (or perhaps multiple styles)” [Monroe et al 1997].
Architectural styles for SoS are discussed further in Section 2.3.2. We build on the principle of general architectural styles to develop the notion of architectural patterns for SoS in Section 4.3.

**Architecture and agile development**

An area that attracts much attention currently concerns questions of how to integrate a well-architected system that benefits from using a coherent architectural style or design pattern (see below) with an agile development process. Nord & Tomayko report “Many practitioners, particularly of agile methods, tend to view software architecture in light of the plan-driven side of the spectrum. They think that architecture-centric methods are too much work, equating them with high-ceremony processes emphasizing document production” [Nord&Tomayko2006]. Agile methodologies may involve beginning code modules and creating initial functionalities without having developed a high-level design or structure for the finished system. This means that, in many cases, an architectural style has not been selected and does not act as a blueprint for future development.

Although some of the benefits of following one or more architectural styles during design and implementation may be unavailable, agile and iterative development processes do bring advantages. The short development cycles, frequent deployment and flexibility of an agile working method allows the development team to respond quickly to user requests and changes (which are generated as users examine early system versions) and to manage customer expectations. Therefore there is much interest in the development of a method to bring together the advantages of both a consistent architectural structure and also an agile development lifecycle. Suggested methods include:

- incorporating the Attribute-Driven Design (ADD) method into an agile development, as suggested by Nord & Tomayko. This method "emphasises addressing quality attribute requirements explicitly using architectural tactics. The quality attributes shape the architecture’s structure..." [Nord & Tomayko2006].
- Carnegie Mellon University researchers have trialled the Architecture-Centric Development Method using student participants which embeds architecture-centric design and analysis methods into the development process [Lattanze2005].
- Boehm and Turner demonstrate a method for adapting an agile process (such as XP) into a form suitable for developing complex, large systems by incorporating high-level architectural planning approaches, design patterns and architectural solutions [Boehm & Turner2004].

**Component-based software engineering and distributed systems**

Component-based software engineering (CBSE) arises out of a desire to reduce the cost and unpredictability of software development. By reusing components which have already been written, tested and tried out by users, we should be able to reduce the cost (compared to writing all code from scratch) as well as the number of bugs and unforeseen problems that tend to arise during later stages of
implementation and testing, when the software is becoming increasingly complex. The field of CBSE focuses on the notion of components which are reusable, and which “are required to interact with each other in a system architecture” [He et al 2005]. Unfortunately, there is not a consensus on exactly what constitutes a “component”. He et al provide a definition based on properties a component should exhibit [He et al 2005]; although Hasselbring defines it succinctly as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition” [Hasselbring 2002]. Different infrastructures have had different definitions of exactly what a component is [Kozacznski & Booch 1998]. Object-oriented software is designed to separate functions into separate modules, where data and functions on that data are bundled together. However, individual objects are frequently too low-level and implementation-specific to be easily reusable between different systems. As a result many component-based systems assume that a component will be something less specific than a single object.

CBSE has its roots in the development of distributed systems and its emphasis on the creation of independently-developed, distributed and highly-modular code components [Kozacznski & Booch 1998]. This environment has naturally led to the development of infrastructures and middle-ware layers to support interoperating distributed modules, including systems like COM+1 and Corba2. Using middleware layers with standards for interoperability allows the inclusion of separately-developed components.

There is a close relationship between CBSE as a field and software architectures. “Primarily, component-based software engineering is a fundamental approach for structuring systems in a modular way” [Hasselbring 2002]. There are many potential stumbling blocks, however, when developing software using existing components. The development and design of component-based system are complex, and it requires a consideration not just of the “standard” system development process, but also a consideration of what capabilities are supplied by legacy components or by commercial off-the-shelf (COTS) components [Hasselbring 2005]. A significant source of problems in CBSE are due to architectural assumptions that may be made by individual components - “architectural mismatches” [Garlan et al 1995]. Garlan et al found that, independently-developed components being incorporated into a new system may be making erroneous assumptions about [Garlan et al 1995]:

- other components in the system, such as which component has overall control and what services will be available
- the system connectors, such as the type of data that is communicated
- the global architecture, such as the topology of the system
- the construction process, such as the order in which components are instantiated and brought under control

1 http://www.microsoft.com/com/
2 http://www.corba.org/
For these reasons, architectural assumptions – style, topography, etc. – need to be made and described explicitly in component-based systems, to reduce mismatches as far as possible. Software engineers working on a component-based system may reasonably expect to spend time thinking about how to resolve these types of problems.

**Architectural description languages**

One major advantage of adopting an explicit architectural style is that it enables analysis of the system to take place. For this reason formal and semi-formal analysis techniques for reasoning about a particular system structure or recognized architecture attract considerable interest.

An architecture description language (ADL) provides a notation or vocabulary for describing system structure and architecture, and facilitates reasoning about that structure. Typically ADLs include some basic abstractions such as “system”, “component”, “connector” [Payne & Fitzgerald 2010]. A wide range of ADLs are available that explore a variety of aspects of architecture [Shaw 2001]. A selection of ADLs, including some of those discussed here, were surveyed in COMPASS deliverable D 22.1 [D22.1 2012]. Well-known ADLs include:

- Architecture Description Markup Language, published by the Open Group. This ADL is based on XML and is designed to encourage the high-level sharing and publication of architectural models [Kruchten et al 2006].
- Acme [Garlan et al 1997, Garlan et al 2003], a generic language that allows a modeller to represent components, connectors, ports, roles and systems [Payne & Fitzgerald 2010]. The accompanying AcmeStudio toolset is an Eclipse-based tool. Syntaxes are defined for graphical or textual representation of models in Acme; Acme has a weak syntax, although annotations on the models (“properties”) do not have a defined syntax and therefore offer a possible means of extending the language.
- Darwin [Magee et al 1995], which focuses on distributed component-based systems. As with Acme, models can be defined graphically or textually, and components can be composed from subcomponents. An accompanying operational semantics is given using π-calculus. Unlike Acme, Darwin supports the modelling of dynamic system reconfiguration, either via a pre-determined operation, invoked at run-time (“lazy instantiation”) or arbitrarily (“dynamic instantiation”).
- Wright [Allen & Garlan 1998] is a formal ADL which relies on a formally specified system architecture [Payne & Fitzgerald 2010]. Like Acme, it uses three structural entities: system, components and connectors. The process algebra CSP is used to define the semantics of architectural models. It does not support hierarchical systems [Payne & Fitzgerald 2010] and concentrates on checking properties of connections and system communications, specifically checking whether protocols of communication of consistent [Allen & Garlan 1998].
- C2 [Medvidovic et al 1996, Oreizy et al 1998, Oreizy & Taylor 1998] is an architectural style which is associated with C2 SADL, an ADL for defining architectures which implement the C2 style. C2 systems are structured around components, with constraints determining how they may be
connected; components are aware only of the components to which they are directly connected, and not those to which they are only indirectly connected. For this reason C2 systems are not really general systems - they adhere to a specific architectural style.

- The Unified Modelling Language (UML) may be used to describe system architecture (via structural diagrams), although it does have relatively weak semantics [Payne & Fitzgerald 2010] which limits its use, particularly for supporting formal analysis of models. Some groups have defined formal semantics for subsets of UML - for example, a subset of Executable UML (xUML) has been translated into mCRL2 process algebra [Hansen et al 2009] and also to CSP||B [Turner et al 2008].

- The Architecture Analysis and Design Language (AADL, formerly known as the Avionics Architecture Description Language) is a description language targeted at complex real time and embedded systems [Feiler et al 2006]. It supports modelling at both the systems level and the SoS level. Models can be represented in AADL in text, in diagrams or as XML. Entities in this language may be categorised as software (a thread, a thread group, a process, data or subprogram), hardware (processor, memory, device or bus) or composite (a system). Property sets may be used to define system-specific properties. Annexes may be used to create extensions to the AADL notation.

- SysML is a modelling language for system specification which may also be used to describe architecture. SysML is discussed in detail elsewhere in this document.

### 2.3. Existing System of Systems-level Architectural Techniques

This section considers a number of sources to investigate architectural techniques as they apply to systems of systems. Two key information sources are considered:

- Relevant academic sources.

The key principles and techniques regarding architectural techniques as they are applied to a SoS are identified from each information source. These principles and techniques, along with the fundamental definitions identified in Section 2.1 and the system-level techniques identified in the previous section will be compared, contrasted and combined into a coherent set in Section 2.4.

#### 2.3.1. DoD ‘Systems Engineering Guide for Systems of Systems’

The ‘Systems Engineering Guide for Systems of Systems’ (referred to hereafter as *The Guide*) published in 2008 on behalf of the Office of the Under Secretary of
Defense (Acquisition, Technology and Logistics) [DoD 2008] has the stated aim of providing “today’s systems engineering practitioners with well grounded, practical guidance on what to expect as they work in today’s increasingly complex systems environment and tackle the challenges of systems of systems.” In the quotations given in this section, the page numbers refer to those of [DoD 2008].

*The Guide* defines an architecture thus {p19}:

- “An architecture is the structure of components, their relationships, and the principles and guidelines governing their design evolution over time [IEEE Std 610.12 and DoDAF]. The architecture of an SoS is a persistent technical framework for governing the evolution of an SoS over time.”

The main aims of an SoS architecture are:

- Address the “concept of operations for the SoS” {p19}
- Encompass “the functions, relationships, and dependencies of constituent systems” {p19}
- Describe “end-to-end functionality and data flow as well as communications” {p19}
- Provide the “technical framework for assessing changes needed in systems or other options for addressing requirements” {p19}
- Provide “an integrated view of the ensemble of systems within the SoS” {p30}

A SoS architecture is not about the constituent systems, but about how they work together. It “does not address the details of the individual systems; rather, it defines the way the systems work together to meet user needs and addresses the implementation of individual systems only when the functionality is key to crosscutting issues of the SoS.” {p47}

The following types of architecture are mentioned in *The Guide*:

- Physical architecture. This “defines the physical components (constituent systems) of which the SoS will be composed.” {p48}
- Functional architecture. This describes the “functionality that the individual systems contribute to the SoS”, providing a functional ‘picture’ of the system and detailing the “complete set of functions to be performed within the SoS as well as the relationships among the functions”. {p48}

*The Guide* suggests that SoS architectures “based on open systems and loose coupling” are an advantage. Such an architecture {p23}:

- “impinges on the systems as little as possible”
• “provides systems maximum flexibility to address changing needs of original users”
• “permits engineers to apply technology best suited to those needs without an impact on the SoS”

This has an impact on the way that trades would be conducted for such an open, loosely-coupled SoS architecture, with trades placing “a greater emphasis on approaches which are extensible, flexible, and persistent over time and which allow the addition or deletion of systems and changes in systems without affecting other systems or the SoS as a whole.” {p23}

Another characteristic of a good architecture is its “ability to persist and provide a useful framework in light of changes” over “multiple increments of SoS development, allowing for change in some areas while providing stability in others.” {p48}

This means that an SoS architecture cannot be static. It must “evolve and mature over time through the result of technical reviews at the SoS level and the linkage to specific systems, as the architecture is employed to increase the capability of the SoS.” {p49} The need for such flexibility is one of the key reasons why open, loosely-coupled architectures are to be preferred, due to the way such architectures limit the impact of change to constituent systems.

“Developing, Evolving and Maintaining an Architecture for the SoS” is seen as one of the seven core elements of SoS systems engineering (SoS SE), with strong links to the other six core elements. The architecture is {p19}:

• Developed from the “existing or de facto architecture of the SoS”
• Considered as an “overlay for the SoS”
• Developed once “an SoS systems engineer has clarified the high-level technical objectives of the SoS, identified the systems that are key to SoS objectives, and defined the current performance of the SoS”.

The Guide acknowledges that, while in the development of a new system “the systems engineer can begin with a fresh, unencumbered approach to architecture”, most constituent systems are already in place and therefore “the SoS systems engineer needs to consider the current state and plans of the individual systems as important factors in developing an architecture for the SoS.” The importance of options and trades in the development of the architecture is recognised, as is the need for the systems engineer to provide feedback “when there are barriers to achieving balance between the SoS and system’s needs and constraints.” {p19}

Architectures are also seen as a key enabler when addressing SoS requirements and solution options. A well-developed SoS architecture:

• Provides “the persistent framework for identifying and assessing design alternatives” {p20}
• Provides “stability as different requirements emerge” {p20}
• Moderates “the impact of changes in one area on other parts of the SoS.” {p20}
• “frames and supports design changes to the SoS over time” {p30}

Developing an architecture is not an optional activity when developing a SoS since “The design of an SoS consists of the architecture of the SoS together with changes to the designs of the constituent systems that enable them to work together according to the architecture.” {p20; editor's italics}

The core element of “Developing, Evolving and Maintaining an Architecture for the SoS” is a key strand that runs across ten of the 16 systems engineering processes defined in the DoD Defense Acquisition Guidebook (DAG), namely {p25; pp51-54}:

• Requirements Development
• Logical Analysis
• Design Solution
• Decision Analysis
• Technical Planning
• Requirements Management
• Risk Management
• Configuration Management
• Data Management
• Interface Management

*The Guide* describes the relationships between these processes and the core elements in detail, and does so both from the point of view of the core elements and from that of the processes. See Chapter 4 and Annex A of [DoD 2008] for details.

Modelling and simulation are seen as key to developing robust architectures, helping to provide a basis for architectural decision and can:

• “support analysis of architecture approaches and alternative” {p10}
• Help “to identify scalability issues or knees in the curve (e.g., concerning requirements or usage assumptions, assumed network bandwidth, or others) beyond which performance starts to break down” {p48}
• Provide “a basis for both selecting an architecture and assessing it over time” {p48}

### 2.3.2. Academic search

As described in the previous section, SoS architectural approaches need to cope with some specific behaviours, such as continuous evolution. Other properties
typically associated with SoS that present challenges to the software architect include:

- **Emerging behaviours.** The SoS exhibits functionality which is not present in any single one of the individual constituent systems; the constituents interact to create potential new functionality. However, accurately predicting and accounting for all the possible emergent behaviours that may be seen is usually prohibitively time-consuming, or is not possible due to lack of information disclosure by constituent systems which have commercial reasons to restrict data access.

- **Very long lifecycles and the presence of legacy components.** Lifecycles of constituent systems are likely to be managed using different methodologies and processes, and are highly unlikely to be synchronised. Some constituent systems may be unable or unmotivated to make adaptive changes that would enable the optimal SoS architecture.

- **The presence of COTS-based systems and independently managed components.** Constituent systems will have other pressures to evolve outside the SoS, meaning that sometimes force the SoS to adapt.

- **A high degree of technical and managerial complexity.**

- **A lack of a central decision-making authority.**

- **Blurred system boundaries.** System boundaries for a single-level system are usually relatively easy to define. By its nature, however, the SoS tolerates the inclusion of third-party, independent constituent systems. Independent systems which would normally form the given operational environment for a single system could be considered as either the SoS environment, or as constituent systems.

- **The multi-disciplinary, cross-domain nature of SoS, which makes misunderstandings and functional gaps more likely.**

- **The inclusion of socio-technical issues.** The users and people interacting within an SoS may be playing the role of the “glue” enabling constituent systems and domains to interact. Their behaviour – which is naturally unpredictable - needs to be taken into account when modelling and designing an SoS.

- **Issues related to trust between independently created constituent systems, and commercial restrictions that often impede information disclosure between them.**

In particular, independence of components, collaboration rather than a central authority, long lifecycles, complexity and evolution have resulted in an emphasis on the need for standards and/or frameworks to ensure that data can be exchanged and that new components can interoperate with legacy components, or with third parties that they may not completely trust [Selberg2008].

There are few published case studies in the field of SoS architecture. Some published studies have described techniques adapted from traditional systems-level engineering which have some common factors with SoSE. Some architectural techniques which specifically deal with SoS, or which are relevant to SoS, include the following:

- **architectural styles relevant for SoS**
- architectural frameworks and component-based software engineering for SoS
- enterprise architectural techniques
- design by contract
- architectural description languages (ADLs) for SoS
- dynamic reconfiguration techniques

**Architectural styles for SoS**

In Section 2.2.5 some well-known architectural styles commonly employed at systems-level are presented. A range of architectural styles have been suggested for distributed systems (for example, see styles described in [Weir 2004]). Although they are not all equally well suited for the particular environments of SoSE, many can be adapted. An architectural style suitable for SoSE should support easy substitution of components; constituent components should be loosely coupled and able to adapt to the replacement of one of their peers. The architecture must be tolerant of the fact that the constituent systems are independent and will evolve over time at a rate that is independent from the evolution of other constituents.

A service-oriented architecture (SOA) is an example of an architectural style which can be considered as an approach for implementing SoS engineering. An SOA architectural model casts constituent components in the role of service provider and service consumer (constituents may be consumers at times, and providers at other times). A service broker component allows constituents to discover the existence of usable services [Arsanjani, 2004]. It is therefore easy to add or remove constituent systems, by ensuring that up-to-date information is available via the broker. An SOA architectural pattern is discussed in Section 4.3.1.1.

Architectural patterns for SoS have not been published previously, to our knowledge. As a growing area, awareness of SoS architectural needs is growing and work in this area is currently underway (e.g., see Kalawsky et al 2013).

**Architectural frameworks and component-based software engineering for SoS**

A key feature of an SoS is the heterogeneity of its constituent systems. With each component being a system in its own right that has an operational existence outside the SoS, many will already have developed their own architectural models and styles, which are likely to differ from those employed by their constituent system colleagues. Consequently, there is an emphasis in SoS architectural techniques on standards and reference architectures which permit modelling and reasoning about heterogeneous components. Many of the component-based software engineering (CBSE) techniques described in 2.2.5 are relevant here. Most research and practice of CBSE architectures and designs so far have been concentrated on single systems. However, there are clearly some lessons that can be leveraged from systems-level CBSE for SoS engineering. As discussed in Section 2.2.5, CBSE at the systems level emphasises standards and frameworks that enable disparate components to communicate and the easy
substitution of one component for another, features strongly associated with SoSE. CBSE when applied to SoS is sometimes known as Component-Based SoS (CBSoS) and a reference framework based on CBSoS is proposed [Loiret et al 2011]. The framework supports the typically heterogeneous architecture found in SoS engineering, allowing heterogeneous architecture description processing and code generation. The reference framework advocates a homogeneous design methodology, with application, middleware and domain-specific services all designing and implemented in a versatile component model. The model supports generic architecture designs, with semantics, roles, domain-specific properties and meta-data that can be specialised via annotations. A toolset is implemented with the component model.

A domain-specific development infrastructure (DSDI) approach proposed by Edwards & Medvidovic [Edwards & Medvidovic 2008] is aimed at the design of large-scale distributed architectures. Although not intended specifically for SoS architecture, it does handle heterogeneous architecture description processing as well as supporting analysis and code generation [Loiret et al 2011]. The approach integrates a domain-specific reference architecture, a domain-specific middleware platform and some domain-specific analysis technologies (e.g., fault-tree analysis in a safety-critical system [Loiret et al 2011]). Integrating these features “allows architects to describe their systems in terms of design elements defined by the reference architecture” [Edwards & Medvidovic 2008]. This is one possible approach for coping with heterogeneity.

**Enterprise architectural techniques**

A system of systems is typically multi-disciplinary and encompasses a range of socio-technical issues. Enterprise architectural techniques may therefore be relevant for SoS modelling in general. These techniques are the result of a recent trend of merging systems engineering and enterprise engineering practices. Enterprise engineering has been defined as “applying holistic thinking to conceptually design, evaluate and select a preferred structure for a future state enterprise to realise its value proposition and desired behaviours” [Nightingale & Rhodes 2007]. The new approach of merging the two “involves applying the principles of systems engineering to the enterprise itself, as a complex entity including the product system(s)” [Rhodes et al 2009].

Rhodes et al identify some key SoS challenges in enterprise architectures as: adding or removing constituent systems; changes in the socio-technical environment; and shifts in the enterprise profile. They propose an epoch-based analysis method for evaluating enterprise architectures in a changing environment, an approach which can be useful for coping with the continuous evolution that is a typical characteristic of an SoS. Using this technique, system lifespan is represented by a series of “epochs”, each of which represents a period when system needs and context are stable. A change in context or needs results in a new epoch. This approach “provides insight into decisions, for example, when in the evolution of the SoS new constituent systems should be added, and when investments should be made in new technology”. The epoch-based analysis begins with defining potential epochs and approximate durations. Rather than a traditional, systems-level approach to architecture where the system moves
towards some vision of how the system should look at a single future point in time, epoch-based architecture approaches encourages thinking about the environments of the SoS.

**Design by contract**

Payne & Fitzgerald explain that “contracts are descriptions of the constituent systems of a SoS given in terms of their expectations and the obligations placed on their behaviour”. The Design by Contract principle was introduced by Meyer [Meyer 1992] as a means of formalising interactions between components, where one provides a service to be consumed by the other. In the formal methods communities, the principle of contracts was investigated even earlier, including Hoare logic triples [Hoare 1969] and the rely-guarantee principle [Jones 1983]. Preconditions state explicitly the conditions that must be satisfied by the consumer in order to consume the service and guarantee a result; post-conditions state explicitly what properties will hold true regarding the output state, assuming that preconditions were met. Invariants make assertions that should hold true about states. The contract therefore formally guarantees that, “given a state and inputs which satisfy the precondition, the operation will terminate and will return a result that satisfies the post-condition and respects any required invariant properties” [Payne & Fitzgerald 2010].

An architecture based on contracts is particularly relevant to SoS engineering because:

- Constituent systems can easily be substituted, since any operation can be replaced by a similar operation as long as it has weaker or equivalent preconditions and stronger or equivalent post-conditions [Payne & Fitzgerald 2010].

- Constituent systems are able to evaluate and make decisions about the reliability of services before employing them, because details are provided about what a service will do [Beugnard et al 1999]. This is particularly useful within an SoS, where constituents which are independent may not have sufficient information to place trust in their equally independent peers.

- SoS designers have the ability to define expected properties on interface contracts for constituent systems. This means that designers can take greater confidence that services will adhere to the desired properties; expectations about services are made clear to constituent system developers [Payne & Fitzgerald 2010].

- If sufficient detail is available, contracts can facilitate dynamic reconfiguration, so that the SoS can react to environmental or internal changes [Beugnard et al 1999]. The scale and complexity of systems of systems, their typical geographic distribution and the independence of their components leads to challenging operating environments, and an SoS typically cannot be rebooted easily if performance is compromised. Therefore dynamic adaptation to changes in the operating environment to cope with problems is an important principle. Unlike a single system, an SoS can tolerate the inclusion of independent, third-party components within its boundaries, and for this reason the boundary between the SoS
and its environment is particularly difficult to define; it may not be clear what is to be considered part of the environment and what is to be considered an independent constituent system.

- The contract system supports analysis and reasoning about the SoS-level properties [Payne & Fitzgerald].

Whilst Meyer focuses on functionality, others have extended the principle of contract-based design. For example, Beugnard et al [Beugnard et al 1999] apply the principle to architectures in which components provide services [Payne & Fitzgerald 2010] in a layered approach. The “basic” and “behavioural” layers proposed by Beugnard et al echo the contracts proposed by Meyer. Beugnard et al, however, also propose two further layers, which provide for contracts that formally specify synchronisation behaviour on operations and quality of service properties that can be expected. Specifying such details enables dynamic adaptation in response to environmental changes [Beugnard et al 1999].

Payne & Fitzgerald [Payne & Fitzgerald 2011] define SysML\textsubscript{C} as a notation for specifying contract-based interfaces. SysML\textsubscript{C} is an extension of the SysML language, and supports the integration of functional and non-functional properties (such as those related to performance). The extension to SysML adds three new features: contract-based interfaces, contract agreements and non-functional properties. Payne & Fitzgerald build on the rely/guarantee model [Jones 1983] with the SysML\textsubscript{C} language, by introducing shared variables on interfaces and the rely-guarantee notation on operations [Payne & Fitzgerald 2011]. “Rely conditions state assumptions about interference on shared variables during the execution of operations by the system’s environment. Guarantee conditions state the behaviour of the operation on shared variables during execution” [Payne & Fitzgerald 2010].

Design by contract is discussed further in Section 4.4.1.10.

**Architectural Description Languages (ADLs) for SoS**

ADLs for the system-level architecture are described in Section 2.2.5. ADLs suitable for SoS architectural development, however, need to address a number of additional issues, such as: the independence of constituent systems; the fact that they evolve at different rates; different architectures between constituents; and the scale and complexity of the SoS itself. As described in Section 2.2.5, different ADLs concentrate on describing different architectural concerns and/or problem domains, which are likely to vary between the constituent systems of an SoS, such that different constituent systems are likely to have different preferred or appropriate ADLs.

One ADL that does support SoS representation is AADL (described in 2.2.5), which allows for representation at either system level or SoS level. AADL provides for three categories of entities: hardware, software or composite (or system). “System components can consist of other systems as well as of software or hardware components” [Feiler et al 2006]. The possibility of creating hierarchical systems which contain other systems allows the language to describe aspects of SoS architectures.
For SoS modelling, it is useful to be able to support a wide variety of concerns and/or domains [Leclercq et al 2007], due to the diversity of the constituent components. Many ADLs suitable for systems-level architecture can be described as monolithic: “Their feature sets and grammar are fixed, and adding new constructs to a monolithic grammar is not possible without modifications to the toolset supporting that ADL” [Dashofy et al 2002]. For this reason, in recent years there has been an interest in the development of ADLs which support diverse semantics for components or design elements (i.e., which support analysis and/or code generation from descriptions in different languages), useful for coping with independent constituent systems and the need to interoperate with legacy components.

- **ACME (described in Section 2.2.5)**, which is an architecture description language for exchange between different ADLs [Loiret et al 2011]. It also supports extensions. However, it does have some drawbacks when we turn to the modelling of an SoS with heterogeneous constituents; for example, it is not supported by a metalanguage that allows description of properties [Dashofy et al 2005]. Further ADLs are available that develop ACME further. For example, ADML, from the Open Group, translates ACME into an XML DTD, whilst xACME is a further variation, also based on XML schemas [Leclercq et al 2007].

- **xADL [Dashofy et al 2002, Dashofy et al 2005]** is an extensible infrastructure which allows architects to rapidly and easily develop new ADLs, based on the principle that most ADLs share a set of fundamental modelling concepts, each supplemented by a small number of specialist features. Using the xADL infrastructure architects can create new ADLs or adapt an existing ADL to suit their specific needs. The infrastructure employs modular and composable XML schemas. xADL focusses on system structure at the design stage, providing support for system evolution as well as the creation and evolution of product families [Leclercq et al 2007]. Although xADL could provide a method for describing an SoS, however, it is not yet supported by an appropriate toolset [Loiret et al 2011] (tools currently available include generic parsers and syntax checkers).

- A toolset proposed by Leclercq et al [Leclercq et al 2007] supports heterogeneous architecture descriptions and is easily extendable. Unlike many other ADLs proposed at the single system level, it is a modular toolset rather than one that implements a monolithic function [Leclercq et al 2007]. The toolset accepts descriptions in various ADL languages, including legacy and domain-specific ADLs.

**Dynamic reconfiguration techniques**

Many modern systems need to exploit “dynamically-formed, task-specific, coalitions of distributed autonomous resources” [Garlan 2000]. For example, services and components available over the internet can be employed temporarily on an as-needed basis, or the system may need to adapt quickly to changes in the operating environment. It is possible, for example, that hardware or software components may become unavailable during the course of carrying
out some particular series of actions. For this reason there is currently considerable interest in the research community in facilitating systems that meld best practice in architectural design with support for system reconfiguration. Whilst much research in the area is not explicitly directly at SoS level architectures, this is clearly an area of interest to many SoS architects. The scale and complexity of systems of systems, their typical geographic distribution and the independence of their components leads to challenging operating environments, and an SoS typically cannot be rebooted easily if performance is compromised. Therefore dynamic adaptation to changes in the operating environment to cope with problems is an important principle. As mentioned previously one method for coping is to ensure that compromised constituent systems can be readily substituted for some equivalent system. A key issue in this area is that of ensuring that dynamic reconfiguration does not compromise the system’s adherence to a recognisable architectural style (see, for example, [Georgiadias et al 2002]).

Whilst reconfiguration can be implemented at design-time, there is a call for systems that can be reconfigured dynamically at run-time to take advantage of services available; reconfiguration may be implemented at a low level, “a reconfiguration of software processes or hardware”, or at a high level, by reconfiguring the architectural topology [Payne 2012]. Reconfiguration is related to autonomic computing, a term introduced by IBM in 2001 as a term to describe a system which is able to “manage” itself in some way. Self-management may rely on a variety of activities, including: self-configuration; self-monitoring; self-healing; self-adaptation; and self-organisation. Self-organising occurs when components collaborate to collectively determine necessary configuration changes and take decisions cooperatively. There is no central authority and no global representation of architecture; typically components need to take actions depending on their individual sensed environment. In contrast, a self-adaptive system uses a centralised approach, relying on a framework of external services and a central internal representation of the system architecture. The current availability of components can be determined by, for example, issuing probe signals and analysing resulting responses or timeouts. A self-healing system will attempt to identify the source of an undesirable change in the operating environment and take one or more actions to ensure that problem can be avoided or removed. For example, network timeouts may be caused by a particular router or server; the self-healing system may be able to reboot or replace a faulty component within its borders, or finding an alternative way to work around the faulty component [Kephart 2005]. Identifying the location of a problem is a key issue in self-healing systems; this can be a non-trivial task in complex, highly-interdependent systems [Kephart 2005]. Another key issue in self-healing systems is the problem of identifying an appropriate action to deal with a problem. One approach is to compile a database of issues which have been seen before (or anticipated) and details of actions that are recommended or have been successful in the past [Kephart 2005].

A challenge in the area of autonomous computing is that of co-ordinating very disparate types of system elements as they monitor the behaviour of themselves
and their neighbours, and select appropriate responses. For example, database components, routers, servers, storage components, workload management and workflows etc., all have separate criteria and metrics for assessing and comparing performance and for making optimisations [Kephart 2005]. Guidelines and standards for benchmarking are therefore a key output from autonomic computing [Kephart 2005].

2.4. Abstraction of Definitions and Principles

The definitions and principles identified in Sections 2.1, 2.2 and 2.3 are abstracted here and summarised.

2.4.1. Abstraction of Definitions

The following ontology covering the concepts related to architecture and architectural frameworks is abstracted from the information sources discussed in Section 2.1 above and defines the terms and concepts that are used in this document.

The concepts shown on Figure 7 are defined as follows:

- **Architectural Framework** - a defined set of Viewpoints and an Ontology. The Architectural Framework is used to structure an Architecture from the point of view of a specific industry, stakeholder role set, or organisation. The Architectural Framework is defined so that it meets the needs defined in its Architectural Framework Context.

- **Architectural Framework Context** - defines the needs (also known in standards such as ISO42010 as concerns) that an Architectural Framework has to address.
• **Ontology** - an element of an *Architectural Framework* that defines all the concepts and terms (*Ontology Elements*) that relate to any *Architecture* structured according to the Architectural Framework.

• **Ontology Element** - the concepts that make up an *Ontology*. Ontology Elements can be related to each other and are used in the definition of each *Viewpoint* (through the *Viewpoint Elements* that make up a Viewpoint).

• **Viewpoint** - a definition of the structure and content of a *View*. The content and structure of a Viewpoint uses the concepts and terms from the *Ontology* via the *Viewpoint Elements* that make up the Viewpoint. Each Viewpoint is defined so that it meets the needs defined in its *Viewpoint Context*.

• **Viewpoint Context** - defines the needs (also known in standards such as ISO42010 as *concerns*) that a *Viewpoint* has to address.

• **Viewpoint Element** - the elements that make up a *Viewpoint*. Each Viewpoint Element must correspond to an *Ontology Element* from the *Ontology* that is part of the *Architectural Framework*.

• **Architecture** - a description of a *System*, made up of a number of *Views*. Related Views can be collected together into *Perspectives*.

• **View** - the visualisation of part of the *Architecture* of a *System*, that conforms to the structure and content defined in a *Viewpoint*. A View is made up of a number of *View Elements*.

• **View Element** - the elements that make up a *View*. Each View Element visualises a *Viewpoint Element* that makes up the *Viewpoint* to which the View, on which the View Element appears, conforms.

• **Perspective** - a collection of *Views* (and hence also their defining *Viewpoints*) that are related by their *purpose*. That is, Views which address the same architectural *needs*, rather than being related in some other way, such as by mode of visualisation, for example.

• **Rule** - a construct that constrains the *Architectural Framework* (and hence the resulting *Architecture*) in some way, for example by defining minimum required *Viewpoints*.

• **System** - set of interacting elements organised to satisfy one or more needs. The artefact being engineered that the *Architecture* describes.

It is important to note here that an architecture is simply considered to be a description of a system, represented by a number of views that are created according to a number of predefined viewpoints from a given architectural framework.

The architectural framework that is used in the production of a system architecture may be an existing framework (such as MODAF or TRAK) or may be a framework created specifically for a particular project. (This latter concept is

2.4.2. Abstraction of Principles

The principles identified for systems and SoSs in Sections 2.2 and 2.3 are abstracted here.

2.4.2.1. From MDA

- *Models* are central to the successful development of a system.
- A number of *models* may be required to fully define a system.
- Models should be produced at a particular *level of abstraction*. In the MDA, the PIM is at the highest level of abstraction, the PSMs at an intermediate level and the code at the lowest level.
- System development involves the *refinement* of models from high to low levels of abstraction.
- (Semi-)automatic *transformation* of one model to another is a powerful technique for creating consistent models represented using different languages that allow different analysis techniques to be applied.

2.4.2.2. From Dickerson & Mavris

- Modelling is seen as essential for the production of architectures
- Multiple architectures should be produced: functional, physical and system architectures.
- Mapping between architectures (e.g. functional and physical) should be done at the same level of abstraction.
- Executable architectures are useful to allow system simulations to be performed and architectural problems determined quickly.
- Architectural reuse should be encouraged.
- Architectural frameworks are an important tool in the production of architectures whatever the domain.

2.4.2.3. ISO 42010

- Architecting takes place throughout the life cycle.
- Architectures should be produced according to defined architectural viewpoints codified in an architectural framework.
- Architectures should be produced to address the concerns of stakeholders using relevant architectural viewpoints.
- It is important to know why a particular viewpoint is being used.
- Consistency rules should be defined between the various views produced and the information contained in them.

2.4.2.4. ISO 15288

- It is essential that an architectural design process is defined and followed. This must cover the definition, analysis, evaluation, documentation and maintenance of an architecture.
2.4.2.5. From DoD ‘Systems Engineering Guide for Systems of Systems’

- Architectures are core to SoS systems engineering and is not an optional activity. Development, evolution and maintenance of an architecture is a key part of Requirements Development, Logical Analysis, Design Solution, Decision Analysis, Technical Planning, Requirements Management, Risk Management, Configuration Management, Data Management, Interface Management.
- SoS architecture is not about the constituent systems, but about how they work together.
- As well as structure, architectures also describe communications, functionality and data flow.
- Open systems and loose coupling should be adopted whenever possible.
- Architectures must evolve over time.
- Modelling and simulation are key to the development of robust architectures.

2.4.2.6. Academia

- Architectural styles (high-level structural patterns) are useful for aiding in reasoning, understanding, reuse and interoperability.
- Architectural Description Languages (ADLs) should be used for describing system structure and architecture and facilitate reasoning about that structure.
- ADLs suitable for SoS architectural development need to address: the independence of constituent systems; the fact that constituent systems are typically at different stages in development; and the scale and complexity of the SoS itself.
- Architectures should support design by contract.
- SoS architectures should support dynamic reconfiguration of constituent systems.

The above abstracted principles are combined and summarised in the following section.

2.5. Summary of Principles

Taking the abstraction of points detailed above, the following key principles relating to the development of architectures can be identified:

- Architectures are core to systems and SoS engineering; production of architectures is not an optional activity.
- Architecturing takes place throughout the system life cycle and resulting architectures must evolve over time.
- Architectures should be produced according to defined architectural viewpoints codified in an architectural framework that includes
consistency rules defined between the various views produced and the information contained in them.

- Architectures should be produced to address the concerns of stakeholders using relevant architectural viewpoints; it is important to know why a particular viewpoint is being used.
- It is essential that an architectural design process is defined and followed. This must cover the definition, analysis, evaluation, documentation and maintenance of an architecture.
- Multiple architectures may be needed (such as: functional, physical and system architectures). These should be produced at a particular level of abstraction.
- Architectures should address both structure and behaviour, including communications, functionality and data flow.
- Mapping between architectures should be done at the same level of abstraction.
- Modelling is essential to the development of architectures.
- Architectural styles and patterns should be used whenever possible, as should open systems and loose coupling. Architectural reuse should be encouraged.
- When used for a SoS, architectures should support design by contract, and dynamic reconfiguration of constituent systems.
- When used for a SoS, architecture is not about the constituent systems, but about how they work together.
3. Classification of Different Types of SoS

This section starts in 3.1 SoS Characteristics with a presentation and discussion of the main characteristics for System of Systems (SoS) as described by several sources. This section sets the stage for the following section 3.2 SoS Application Examples, where application examples of either existing or theoretical application examples of SoS are shortly presented and discussed in relation to SoS types and recognized architectures. Section 3.2 presents examples from different application domains with the purpose of giving the reader an understanding of different types of SoS and also to challenge the use of different architecture styles and patterns in relation to these application examples. Section 3.3 Relations to SoS Architectural Patterns will discuss the common architecture findings in relation to the architecture styles and patterns presented in section 4.3 Architectural Patterns.

3.1. SoS Characteristics

There is not yet a widely accepted single definition for SoS. However, a large number of researchers have attempted to document the characteristics of SoS. The most well-known of these has been published by Maier [Maier 1998]. Maier describes five properties which are exhibited by SoSs as opposed to single systems. These properties have often been treated as a means of defining an SoS; using this system, a system exhibiting all the five properties would be considered an SoS, and would not be considered an SoS if it does not exhibit all five.

The five properties are:

- “Operational Independence of the Elements: If the system-of-systems is disassembled into its component systems the component systems must be able to usefully operate independently” [Maier 1998].
- "Managerial Independence of the Elements: The component systems not only can operate independently, they do operate independently. The component systems... maintain a continuing operational existence independent of the system-of-systems” [Maier 1998].
- “Evolutionary Development : The system-of-systems does not appear fully formed. Its development and existence is evolutionary with functions and purposes added, removed, and modified with experience” [Maier 1998].
- “Emergent Behavior: The system performs functions and carries out purposes that do not reside in any component system. These behaviors are emergent properties of the entire system-of-systems and cannot be localized to any component system. The principal purposes of the systems-of-systems are fulfilled by these behaviours” [Maier 1998].
“Geographic Distribution: ...the components can readily exchange only information and not substantial quantities of mass or energy” [Maier 1998].

The general concepts of independence, evolution and emergence have been quite widely accepted, and variations of these properties feature in subsequent definitions of SoS. However, although many SoSs will exhibit a geographically distributed architecture simply by their nature, not all researchers accept that this is a defining and required feature of an SoS. For example, Cocks questions “If geographic proximity were truly a discriminator, then the question must be addressed: ‘How close is too close to still constitute a system of systems?’” [Cocks 2006].

Fisher [Fisher 2006] builds on Maier’s properties, emphasising that “those characteristics of systems of systems derive from the operational and managerial independence of their constituent parts, from independent evolution, and from the character of emergent effects” [Fisher 2006]. Fisher examines the question of how management independence, evolutionary independence, and emergent behaviour arise in an SoS, concluding that “all of these characteristics derive from the presence of autonomous constituents in the system... The presence of autonomous constituents is both necessary and sufficient to characterize systems of systems... By autonomous we mean that an entity can exercise independent action or decision making” [Fisher 2006]. Fisher argues that the geographic separation of components gives rise to the presence of autonomous components, which in turn results in independence of operations, independent management, independent evolution and emergent behaviour that Maier documents. The results of autonomous constituents (operational independence, managerial independence, and emergent behaviour) are therefore the unique characteristic of systems of systems according to Fisher.

An alternative set of the properties that distinguish an SoS from a conventional, single system has been proposed more recently by Boardman & Sauser. Their defining SoS properties are [Boardman & Sauser 2006, Baldwin & Sauser 2009]:

- **Autonomy:** each constituent system must “be free to pursue its purpose” [Boardman & Sauser 2006, Baldwin & Sauser 2009].
- **Belonging:** each constituent system will need to adapt to become part of the SoS and must “be persuaded of the value of all this – to change, to render service, and to collaborate with other systems... belonging does mean partness” [Boardman & Sauser 2006]; “belonging is a reciprocal forfeiting of this independence” [Baldwin & Sauser 2009].
- **Connectivity:** unlike a single system, in an SoS there is a “call for the unravelling of the encapsulated system, giving us access to some of its inner connectivity that does not normally appear at its surface, or system boundary” [Boardman & Sauser 2006]. This is forced on the SoS by the presence of legacy components and the acceptance that
constituent systems determine for themselves how they wish to connect to each other.

- **Diversity**: the SoS “should, out of necessity, be incredibly diverse in its capability as a system compared to the rather limited functionality of a constituent system, limited by design” [Boardman & Sauser 2006]. This diversity can be applied to the variety of connections systems as well as the diversity of the whole.

- **Emergence**: the constituent parts interact together and produce new behaviour that is more than simply the sum of their individual behaviours. “The difference between autonomy and diversity is that diversity ensures multiple system capabilities while autonomy ensures these autonomies can complete the system goal” [Baldwin & Sauser 2009].

Like Maier, Boardman & Sauser emphasise the independence of constituent systems and emergent behaviours. A different definition by Abbott [Abbott 2006] does not explicitly feature these properties, however. Abbott views an SoS as an environment containing the constituent systems rather than a hierarchy of constituent systems. Abbott defines three key characteristics for an SoS, arguing that an SoS should be [Abbott 2006]:

- **Open at the top**: there is no hierarchy and no “top level” system. New constituent systems and applications may be added continually.

- **Open at the bottom**: “there is no fixed bottom level for a system of systems. The lowest level of a system of systems may be changed out from under it at any time” [Abbott 2006].

- **Continually evolving, but slowly**: An SoS is never completed and evolves continually along with its environment.

Although Abbott does not explicitly characterise SoS constituent systems as autonomous, there is in Abbott’s definition an inherent lack of central control over the SoS as a whole, with no hierarchy imposed on the constituents. The presence of a structure imposing hierarchical control, argues Abbott, might prevent beneficial collaborations.

Cocks [Cocks 2006] takes the view that geographical distribution, evolution and emergence are features typically associated with an SoS, simply by its nature, but argues that “they are not discriminating attributes that make a system into an SoS”. Instead, Cocks proposes a simpler definition: “A system of systems is a product of a system engineering process that contains one or more systems for which significant aspects of the integration and life cycle development of the component system(s) are beyond the managerial control or influence of the larger system.” The definition clearly emphasises the independence of components: “It is the managerial control of the systems – rather than any intrinsic characteristic of the systems themselves – that is significant”, clarifies Cocks [Cocks 2006].
The range of discussion of systems of systems does coalesce in some respects, even if few authors use the same terms for related concepts. There’s a general agreement that independence of constituents is an important element; this is something that leads to much of the unique complexity of SoSE. Different researchers focus on different aspects of this, including the autonomy of the constituent systems, the lack of a central control or hierarchy, and constituent systems’ continued existence outside the SoS (which introduces pressures to evolve in ways unrelated to the SoS goals). Emergent behaviour also features in most descriptions or characterisations of an SoS, along with some recognition that an SoS typically sees continuous evolution.

A literature survey conducted by COMPASS participants [Nielsen et al 2013] identified eight key properties of systems of systems, based on a wide examination of the literature characterising SoSs. These eight properties of an SoS are:

- Autonomy: each constituent can make independent decisions
- Independence: constituents continue to operate self-sufficiently outside the SoS
- Distribution: the constituents are scattered, such that effort is required to connect them
- Evolution: the SoS develops throughout its lifetime
- Emergent behaviour: the overall SoS delivers higher functionality than that which is available from any of the constituents separately.
- Dynamic behaviour: constituents can be added or their interrelationships adjusted
- Interdependence: there is a mutual dependency between constituents forming the SoS, which rely on each other in order to fulfil the SoS’s goals.
- Interoperability: the SoS incorporates a wide range of components, integrating and adapting interfaces, protocols and standards to bridge the systems.

These eight properties attempt to summarise the main concepts addressed repeatedly in the literature, even if authors themselves use different terms for similar concepts.

Systems of systems can be separated into four separated categories based on a single widely-used classification system. Three of these categories were originally proposed by Maier [Maier 1998], and the “acknowledged” class was an addition later proposed by Dahmann and Baldwin [Dahmann & Baldwin 2008].

The four categories of SoS are:

- The virtual SoS, which lacks a central control or management and has no commonly-agreed SoS goal, although there are still constituent systems which interact in the process of delivering some emergent functionality [Maier 1998].
- The collaborative SoS, in which constituent systems participate voluntarily to achieve an agreed-upon goal. The internet is a widely-used example of a collaborative SoS, because it relies on manufacturers and software providers voluntarily adhering to a set of standards in order to achieve interoperability [Maier 1998].

- The acknowledged SoS, in which constituents are still independent with their own goals, lifecycles and resources, but in which the SoS also has agreed objectives, a management authority and its own resources. Changes are achieved by collaboration between the needs of the SoS and the constituents [Dahmann & Baldwin 2008].

- The directed SoS, in which an SoS is built and managed in order to achieve a specific goal. There is central management during the initial development and during evolution which might alter functionality or deliver new functions. Constituent systems can operate independently, but their participation in the SoS normally receives some priority [Maier 1998].

These categories are a factor which may be taken into account when making decisions about the requirements engineering, SoS architecture or design, since the level of central managerial influence that may be exerted dictates how collaborative the constituent system is likely to be when working to meet SoS goals, and the likely level of connectivity that can be achieved.

3.2. SoS Application Examples

This section will give examples of system of systems from different domains, with the purpose of making a classification of these and see how they are or could be architected.

3.2.1. COMPASS SoS Case Studies

This section will introduce the two industrial case studies in the COMPASS project and the London Emergency Services SoS, which Newcastle University has worked on previously.

3.2.1.1. Bang & Olufsen Case Study

Description

The Bang & Olufsen (B&O) case study [D42.1 2013, Hallerstede et al 2012] is a SoS for control of connected Audio-Video products. Traditionally, B&O delivered a complete AV solution by using their own proprietary technology to integrate and connect the equipment. But nowadays, with the growing trends of open technologies and interoperability, it is no longer feasible to continue doing so.

B&O now requires that AV systems be composed of various connected standalone, independent products. Each product can be considered to be an independent system. Therefore, the focus must be on interoperability and compatibility between these various products. On the other hand, these AV SoS
must still provide emergent behaviour in the form of new smooth experiences for the user that result from combining the various products (this is called the B&O brand user experience).

**Industrial or theoretical case**
This is a clear example of an industrial case. It is, in fact, one of the industrial case studies of the COMPASS project.

**SoS Type**
The B&O example can be considered to be of Collaborative type. B&O plan to leverage the DLNA standards for interoperability. In this case, each Constituent System (CS) is totally independent from each other. However, all the products voluntarily follow the DLNA standards in order to fulfil the overall goal of providing the end-user with a high quality AV experience.

As for control and evolution of each CS, it is quite clear that they are also totally independent since it’s possible that each AV product is from a different manufacturer which guarantees independence.

**Architecture**

![Figure 8 - Base B&O SoS](image)

An example AV SoS can be seen in Figure 8 - Base B&O SoS. This is a simplification of the actual SoS, as things such as network control and clock domain are not represented. Nonetheless, the figure represents the base architecture from which most AV solutions are derived from.

We can see that it contains multiple different AV products (Play products are a brand for devices such as iPods, etc.). The various products must communicate with each other, to assure the desired emergent behaviour can be provided with any combination of the products. Finally there is a gateway product for the integration of legacy products (any product that does not support the interoperability protocols).
From a pattern standpoint, the SoS could be modelled using the Service Oriented Architecture pattern (see section 4.3.2), where each product exposes its functionality as a series of services compliant with the interoperability protocol.

3.2.1.2. **Insiel Case Study**

*Description*

The case study, produced by Insiel [D41.1 2013], is an emergency response system (ERS) which coordinates and manages resources from emergency systems controlled by independent public 118 Emergency services medics and paramedics and the emergency wards of several hospitals. The SoS is composed of the following constituent systems:

- The "CUS" (*Centrale Unica di Soccorso* = unified rescue central, Unified Emergency Call Centre) is the core system in the SoS
- The *Emergency Response Units (ERUs)*, usually consisting of an ambulance or flycar with crew such as driver and medical staff
- A *Radio System* to provide connectivity of the CUS with ERUs
- A *Phone System* to provide connectivity for incoming phone calls as well as communications with other systems outside the SoS boundary.
- A *Telecommunication system* providing connectivity between the Phone System and external callers.

The emergency response case study may be considered a combination of a directed/acknowledged SoS. The communication systems (radio and phone) are provided by external organisations, and are not necessarily aware of their participation in the SoS. The CUS and ERU systems form the main management of the SoS, with the CUS responsible for the delivery of the emergent SoS behaviour.

*Industrial or theoretical case*

This is another clear example of an industrial case and one of the industrial case studies of the COMPASS project.

*SoS Type*

The ERS is an acknowledged SoS. Insiel own and manage the CUS, however have no control over the other constituent systems. The CUS is considered the ‘manager’ of the SoS. Typically, changes made to the SoS are based upon collaboration between the constituent systems.

*Architecture*

A simplified topological diagram of the ERS is shown in Figure 9. The Phone System and CUS are shown as constituents of a combined Insiel system – however this is just a convenient way of modelling the SoS (the Phone System is a constituent system operated by Insiel, although developed, provided and managed by a separate organisation).
The CUS is connected to its environment through the TeleCom, and to the ERUs through the Radio System. The TeleCom, Phone System and Radio Systems are all explicitly modelled as they may be composed of large numbers of software and hardware components and are sources of faults.

Although it is not clear in this diagram, the ERS could be considered a centralised SoS. The CUS is responsible for using the services provided by the other constituent systems so to ensure the provision of aid to a specified target.

### 3.2.1.3. London Emergency Services

**Description**

The London Emergency Services Liaison Panel (LESLP) major incident procedure manual [LESLP07] summarises procedures and arrangements for the coordination of the London emergency services (including the Metropolitan Police, London Fire Brigade and London Ambulance Service) and other services (National Health Service, Local Authority, etc.) in the event of a major incident. A case study was created in the COMPASS project, which considered a subset of the entities of the LESLP manual and simplified the operational procedures [Payne&12a]. In this document, LESLP refers to this case study.

In this case study, each emergency service detailed may be considered a heterogeneous system. Given the characteristics of a SoS by Maier [Maier98], we consider the collaboration of the emergency services during a major incident to be a SoS. Each service has independent operation and management, the functions and membership of the response and the emergency services change over time, behaviour emerges from the combination of service functions, and the emergency services may be geographically distributed.

The emergency services, and thus the constituent systems are: Police Service, Ambulance Service and Fire Service.

Each constituent system consists of people with an associated rank. The rank of each officer determines the permitted communication channels between officers. This is explored in more detail in [Payne&12b]. Also identified, though not
considered in the case study, are the communication systems used to transmit data between systems.

**Industrial or theoretical case**

The case study presented in [Payne&12a] is academic, however based upon a real world procedure manual developed by LESLP – a group of representatives from the London emergency services, governmental bodies and experts.

**SoS Type**

This simplified case study can be considered an acknowledged SoS. Each of the constituent systems retains its own management, objectives and funding. However, there are recognised SoS-level objectives in a major incident, with a Gold command formed, which manages the strategic decisions of the SoS. Changes made to the SoS – in terms of the composition of each service, and also of the SoS itself (for example adding additional constituent systems in the form of expert systems).

**Architecture**

The LESLP case study in [Payne&12a] considers information flow between officers in the individual systems, and communications between those systems. As such, the architecture was described not in terms of the individual constituents, but in terms of the command levels of those constituents.

Figure 10 shows the permitted connections between the different ranks of the SoS constituents. It should be noted that the figure only considers the connections with respect to the casualty information clearance case study, and not the whole SoS behaviour. As such, for this aspect of the case study, the SoS is considered to be centralised.

The remainder of the architectural model of the case study is given in [Payne&12a].
3.2.2. Industrial Cases

The following sub-sections describe typical architectural styles found in a range of industrial and commercial applications.

3.2.2.1. Health Care Systems

Description

The health care domain is today dominated with a very large number of complex systems, where many of these perform nearly identical functionality. Examples of such systems are Electronic Patient Record (EPR) systems, where for example Denmark has 4-5 different comparable EPR systems running in different areas of the country. These systems are typically running on hospital sites and with current problems in exchanging data, when patients are treated on hospitals in other parts of the country.

![Diagram of Health Care Systems](image)

The health care domain experience new challenges in these years caused for example by introduction of private owned hospitals and in the near future there will be a more close cooperation and communication between hospitals and the local health care systems in the municipalities and with peoples local doctors. Another challenge is in the home care area where there in the coming years will be a huge number of new health care systems as well as care systems installed in private and in nursing homes. A citizen can at the same time be in contact with or registered in one or more hospital systems, a local home care system, the local doctors system, systems at specialist doctors and possible connected to a healthcare device oriented system in the home.

Figure 11 shows the different levels of systems in a national health care domain, where each constituent (CS-System) in practise can be a whole set of systems as for example at a hospital, where a large number of it-systems are working more or less independent but typically collaborating with the local Electronic Patient Record system. Another current trend is in the private sector where there currently are many remote health care research project taking place in private homes. The typical situation here is, that stand alone stovepipe systems are
installed in each home, where each system performs a limited set of measurements on a given person and where each systems has its own duplicated it-infrastructure.

In this way the health care domain is an obvious place for future development of SoS, where existing constituent systems are integrated or transferred to form new SoS. The healthcare domain will in the future be dominated with SoSs, where each SoS has its own set of system goals to achieve and where some of the constituent systems will participate in several other SoSs.

An example of a National Constituent System is the “Shared Medicine Card System” in Denmark (FMK), which has recently been put into operation. The purpose of this system is to have a single and central place in Denmark where all medicine prescriptions for all citizens in Denmark is registered and where all medicine purchases in the future are registered as well. The primary input source is the system at the local doctors’ site, but should be extended to registration of medicine prescription at all other places in the health care system as well. The intention with this system is that is can be accessed by all other systems who needs the latest and valid medicine prescription list for a given person, including the possibility for the citizens to access own data as well by using a normal web-browser. This shared medicine card system is used in hospitals for delivering the right medicine to the patients and in the home care domain to give the right medicine to the clients who gets assistance with the medicine administration.

The shared medicine card system can be categorized as a constituent system which is collaborating and being part of several SoSs as indicated on Figure 12, where SoS1 is the SoS which mainly provides and updates the information in the Shared Medicine Card CS-system, whereas SoS2 is one of the SoS which mainly uses the information provided by the shared medicine card system.

![Figure 12 - A scenario with a constituent system collaboration with two systems of systems](image-url)
In the near future the shared medicine card system could be included in another SoS by collaborating with for example systems at the pharmacies for medicine control and ordering purposes.

**Industrial or theoretical case**
Theoretical case supplemented with the current health care systems situation in Denmark.

**SoS Type**
The shared medicine card SoS with the communication with the local doctors it-systems can be regarded as a directed SoS as this main system is controlled and maintained by a central authority and designed to work with the doctors it-systems. The doctors system is a separate constituent system with other purposes as well and managed locally and maintained by the IT-vendor of the system. The shared medicine card systems more voluntarily cooperation with other constituent systems as for example the hospital systems or the home care systems can be regarded to be forming a collaborative SoS.

**Architecture**
The public interfaces of the medicine card system are implemented by web services a part of a service oriented architecture (see section 4.3.2), which would also be an obvious choice for other national SoS as well.

For the other SoS a Data-Centric Publish Subscribe architecture (see section 4.3.3) could be a valuable alternative as well, but mostly when new SoS are defined from a central or national point of view.

### 3.2.2.2. Finance and Investment Systems

**Description**
The study [Roussos et al 2008] is said to describe a “large scale system”. It is however assumed to be composed of sub-systems that are fairly independent but need to work together to deliver services to the users. Services usually involve several sub-systems. It is essential for the trading and banking activities that the services are consistent, that e.g., different estimates for the same product are calculated by different algorithms.

**Industrial or theoretical case**
The analysis is based on an industrial system. However the implementation appears to remain theoretical.

**SoS Type**
The system appears to be directed with strict security enforced.

**Architecture**
The architecture is service-oriented with all sub-systems connected (conceptually) by a network.
The service-oriented architecture pattern (see Section 4.3.2) is most appropriate for this kind of SoS because the main purpose of the SoS offering services across collections of constituent systems. The publish-subscribe pattern can be used as well for financial systems, but will require a common communication paradigm based on a data-centric publish-subscribe mechanism.

3.2.2.3. Package Shipping and Tracking Systems

Description
This SoS realizes a real-time package tracking system, where each of the constituent systems is a stand-alone package shipping system [Joshe 2013]. Each of these shipping systems has its own method of keeping track of its operations and makes its package status data available in its own format, at different intervals and by different techniques.

The package tracking CS-system can collaborate with other systems who requires tracking information which can be used for example by ordering systems do deliver updates on expected delivery dates or by systems which can track and find lost or delayed packages.

Industrial or theoretical case
This is an industrial case.

SoS Type
The package tracking system is a collaborative system where each constituent shipment system collaborates to deliver package tracking information to the Package Tracking CS-System. In this example each shipping system currently uses its own technique and format to deliver the tracking information which is transformed in the package tracking CS-system to the wanted format and collated with the package
tracking information from the other systems. An enhancement would be if each shipment system would conform to the same standard interface for querying package tracking information.

Each of the shipment CS-System is an independent system with its own purpose, management and control.

**Architecture**

This system can be architected in different ways. The architecture described in [Joshe 2013] is partly based on the Data-Centric Publish-Subscribe architecture style (see section 4.3.3) implemented based on the Data Distribution Service (DDS) Standard, where each shipping system can act as a publisher of package tracking information, which are pushed to the registered subscribers.

Another possible architecture could be using a service oriented architecture (see section 4.3.2), where each shipping system has a service interface for querying package status but this would lead to a more inefficient system as the tracking systems has to poll all the collaborating systems for status information.

### 3.2.2.4. Maritime Transportation Systems

**Description**

A Maritime Transportation Systems deals with transfer of goods and services between national and international destinations [Mansouri et al 2009]. Traditionally the following central concepts are distinguished:

- Ships
- Ports
- Intermodal connects
- Waterways
- Users.

**Industrial or theoretical case**

Industrial case. However SoS potential not exploited to full potential.

**SoS Type**

Collaborative with many stakeholders: civil society, governmental sectors, private sectors

**Architecture**

The architecture follows the five concepts mentioned above where each one is a complex domain, e.g.:

![Figure 15 - Example of a complex sub-system](image)
This SoS is unlikely to have a single architectural pattern. One could even claim that the sub-systems such as Intermodal Connects are themselves Systems-of-Systems.

The service-oriented architecture pattern (see section 4.3.2) is most appropriate for this kind of SoS because the main the SoS offers services across collections of (by themselves very large) constituent systems that can only be loosely coupled.

### 3.2.2.5. Space Exploration Systems

**Description**

The Solar System Mobility Network (SSMN) is a proof of concept of the use of SoS strategies and approaches to the problem of space exploration [Sindiy et al 2007].

SSMN proposes the construction of a system infrastructure across the solar system to support multiple missions and increase humanity's exploration capabilities of the solar system. The key objectives of applying SoS to space exploration are increasing reachability, versatility and sustainability.

Ownership and control is not particularly discussed although there are mentions of several stakeholders and other agents that influence the behaviour of the system such as scientists, manufacturers, NASA, etc. Some of these stakeholders have different goals (such as profitability for private contractor) but they must all act under whatever central authority oversees the SSMN.

When it comes to the evolution, maintenance and management of the CSs, no guidance is offered. For the SSMN simulation model, all CSs were developed and operated in coordination.

**Industrial or theoretical case**

The example is theoretical based on modelling and simulation approaches. In fact SSMN was first conceived as a challenge to be presented at a conference.

**SoS Type**

SSMN can be considered directed. The common and concrete goal of SSMN is to maximize exploration capabilities in the solar system. All the CSs present would be developed and work towards that goal. More importantly though they all operate independently they are working towards the same goal. For example the supply production facilities operate on their own but their purpose is simply to produce materials that can be used by other CSs to further the overall goal.

**Architecture**

The proposed model for SSMN features several different CSs: Human missions to nearby planetary bodies; Robotic missions to all planetary bodies in the solar system; Servicing missions for deployed space based CSs; Supply Production Facilities using available resources to produce fuel, parts, etc; Cargo Transport Missions for the transport of supplies; Manufacturing plants for the construction of spacecraft for deployment across the solar system.
These CSs are distributed across the solar system in accordance to a specific network topology. Some CSs are located on planetary bodies (such as the Earth, Moon, Mars, etc.) while others are artificial satellites and space stations (ex: LEO, seen in the figure), simply floating in orbit. A subsection of this network topology is shown in Figure 16 - SSMN Network topology crop. The Centralised Architecture pattern (see section 4.3.1) might be used to model SSMN. In this case, certain nodes of the network would be chosen as the central systems (for example the Earth station).

### 3.2.2.6. Meteorology Systems

**Description**

The Global Climate Observing System (GCOS) was established in 1992 (thus predating a great deal of SoS research) and is composed of systems and networks from all over the world in order to provide the climate observations (and relevant data) to fulfil various needs including monitoring, research, prediction, impact, etc [WMO2008].

GCOS is sponsored by four international organizations: the World Meteorological Organization, the Intergovernmental Oceanographic Commission of UNESCO, the United Nations Environment Programme and the International Council for Science. The actual running and management of GCOS is in the hands of the GCOS Steering Committee, appointed by all 4 organizations.

Most CSs of GCOS are themselves climate observation system though they are smaller in scope or size. The main CSs are:
• The World Weather Watch Global Observing System and Global Atmosphere Watch
• The Global Ocean Observing System
• The Global Terrestrial Observing Network
• Various Space-based observing systems
• Various smaller in situ observing systems

Most of these systems collect data and GCOS itself is all about collection and exchange of data. Most of the coordination effort in GCOS has to do with what data to collect and how to collect it. In terms of what to collect, GCOS has defined essential climate variables for three domains (atmosphere, ocean, terrestrial). The variables are meant to be technically and economically feasible to capture while also having high value for climate observations. Examples include air temperature (atmosphere), salinity (ocean) and leaf area index (terrestrial). In terms of how to collect said data, there is already a great deal of standardization for climate observation but GCOS has enriched it with a set of Climate Monitoring Principles which guide the design and implementation of CSs.

**Industrial or theoretical case**
This is one of the clearer examples of an industrial case since it is an actual SoS in production and use by various organizations worldwide.

**SoS Type**
GCOS can be considered Acknowledged. All the various CSs are working towards a common goal: gathering climate information. Management and control is independent since the CSs are under ownership of different organizations. The evolution of the SoS itself and its goals is in the hands of the steering committee which should in theory represent the various stakeholders in the entire SoS.

![GCOS Overview](Figure 17 - GCOS Overview)
Architecture
GCOS does not have any special concerns with an overall SoS architecture. Its focus is on data formats and communications. The various CSs must simply adhere and to the GCOS standards and may freely exchange information. It is possible to model GCOS using the Publish-Subscribe pattern (see section 4.3.3).

3.2.2.7. Air Traffic Control Systems

Description
An air traffic system having many stakeholders (airlines, passengers, flight controllers, flight authorities) but control of the entire system has traditionally been in the hands of an air traffic authority: the Air Traffic Service Provider.

The Air Traffic Management Partnership (ATMP) [Ball 1997, Geddes et al 1998] and its proposed model of Free Flight suggest that control of air traffic be split among several agents:
- ATSP: air traffic controllers, airports etc.
- Air Operations Centre (AOC): airlines, their fleets, flights, etc.
- Flight Deck (FD): one for each airplane

The goal of ATMP is to spread the tasks and responsibilities of air traffic control to the various stakeholders. In this SoS we have three types of CSs, one for each agent. The collaboration between all three will allow for significant advantages such as: better resolution of traffic conflicts, better scheduling and much better scalability of the entire SoS (each new player brings with him additional computation power).

Industrial or theoretical case
This is an industrial case example. It was developed by Lockheed Martin, tackles a real world problem and among the chief goals are improvements in productivity and economic savings.

SoS Type
This system can best be described as acknowledged but with several caveats. The SoS has multiple goals but it can generally be said to want to maximize the efficiency of air traffic. All CSs are interested in this shared goal. However, each has their own goals, which may sometimes be at odds with the remaining CSs. For example, the airline operators want to minimize fuel costs (so they would be interested in short taxis to take off). On the other, the airports have other priorities.

Control of the SoS is an interesting aspect since the various players in ATMP have control over their own CS but they can be subjugated to the ATSP. So in a sense, the system could be considered directed since everything must obey the ATSP. However, the various CSs have their own purposes to fulfil and their evolution is not dictated by the ATSP, though it may face some restrictions. Still, the control of each CS can mostly be said to be independent. Perhaps the best way to describe it is as an acknowledged SoS with a hierarchy among its CSs.
Architectures

The various CSs tend to collaborate in pairs. For example, FDs can collaborate with each other (collision avoidance), with the AOC (scheduling) and with the ATS (conflict resolution). As such there is no fixed structure. The various CSs of ATMP work under shared models of purpose.

They exchange information and recommendations on how to deal with various situations. Finally, there is a hierarchy of authority: AOCs can override FDs and ATSPs can override both. None of the patterns described in this report is very appropriate for ATMP. Possible patterns for modelling ATMP would be ones focusing on either the hierarchy of authority or the peer-to-peer communication between CSs.

3.2.2.8. Defence Systems

Description

Warfare is moving away from “dominance of weapons” towards “dominance of information”. Dominance over a future battlefield will only depend to a limited degree on superior weapons; to a large degree it will depend on having the best information about the battlefield and the ability to carry out precision strikes.
This requires a high amount of sensor information about the battlefield that can only be collected by joint forces (Ground, Naval, Air). It is argued that this requires a change in the organisation of the military leading to more centralisation.

**Industrial or theoretical case**
This is a theoretical case that has been evaluated on a smaller scale in local conflicts, e.g., in Ex-Yugoslavia. The case described in the article below attempts to formulate the consequences of introducing a SoS approach on a large scale.

**SoS Type**
A core part of the SoS must be directed, very tightly integrated. This may not need to apply to the system as a whole. Some parts may be integrated more loosely.

**Architecture**
A centre piece of the architecture is the digitized battlefield where all information is collected in order to inform precision strikes. A lot of the requirements concern the necessary changes to the organisation of the military itself. The figure below only indicates the part of the SoS that deals with the information. However this is the part that needs to be tightly integrated.

The centralised architecture pattern (see Section 4.3.1) is most appropriate for this kind of SoS because the main purpose of the SoS is command and control.

### 3.2.2.9. Reconfigurable Unmanned Aircraft Systems

**Description**
Unmanned Aircraft Systems (UAS) has in the past been composed of one Ground Control Station (GCS) controlling one Unmanned Arial Vehicle (UAV) also called a drone. The next generations of these systems will have a many to many relationship between many GCS’s controlling and supervising many UAVs and in this way fulfilling a multiple of mission objectives [Jong 2011]. One of these scenarios can be controlling a swarm of unmanned aircrafts. For controlling and guiding this future development the US Department of Defence published in 2007 a roadmap for Unmanned Systems [DoD 2007]. In these coming systems it is foreseen that any end-point must be able to communicate to any other end-point and be able to meet the
real-time requirements for this communication. One of the challenges in these futures SoS will be how to design UAS’s around a common communication and integration platform supporting the requirements for dynamic adaptability.

![Diagram of Data Oriented Architecture for future UAS System of Systems](image)

**Figure 21 – Data Oriented Architecture for future UAS System of Systems**

**Industrial or theoretical case**
Unmanned Aircraft Systems are actual SoS used in different parts of the world both for surveillance and for attack purposes.

**SoS Type**
The current UAS SoS can best be categorized as directed systems, with a specific purpose and with a centralized control, where the communication is typically build around a connection-oriented paradigm with connections between GCS and each UAV.

**Architecture**
Current UAS systems have a centralized architecture but future UAS will require a more decoupled architecture as the one shown on Figure 21 with support for dynamic re-configurability and being built around a common communication and integration platform. A possible architecture could be to use the Data-Centric Publish-Subscribe pattern (see section 4.3.3).

### 3.2.3. Examples from Compass Interest Group

The SoS examples in this section are obtained from a subset of the COMPASS Interest Group (CIG). As a part of Task 4.1.3, the CIG members have submitted a brief description of example SoS problem areas. The CIG-inspired examples are given here.

#### 3.2.3.1. Smart Grid Systems

**Description**
Grid Manager offers automated and intelligent energy saving and smart grid services [D43.2 2013]. This is done by means of a cloud service infrastructure, integrated and interacting with a continually growing set of third party systems. Several kinds of data form the foundation for the energy services, the most obvious being metering data. Metering data is collected from a wide range of different smart meters, with different communication mediums, as well as being
either pushed or pulled from metering solutions offered by partners. Each metering solution forms is an independent system, running autonomously. The only exception to this is Grid Manager’s own metering solution, which allows for a tighter coupling to the main services. Metering data are derived from all sorts of energy metering, i.e. electricity, water, gas, heating, cooling, but also other data such as temperature or even more loosely defined values like a pulse given for every unit a machine in some factory produces. Apart from metering data there are other kinds of data such as energy tariffs, CO\textsubscript{2} indexes, as well as many custom indexes supplied by customers. Examples of custom indexes are employees, square meters of a facility, or key production numbers.

For most of the constituent systems the communication protocols vary as well as the capabilities. That is, most metering solutions are different. Differences are often in the form of push or pull of measurements, varying resolution of measurements, varying support for actuators (i.e. allowing the SoS to control customer appliances).

**Industrial or theoretical case**
This is an industrial case describing a system that is already in operation.

**SoS Type**
This SoS is collaborative. Furthermore, some systems can be integrated tighter allowing more services to be created. However, this requires dedicated Grid Manager hardware.

**Architecture**
Cloud-based systems as such are usually set up as service-oriented architectures (see Section 4.3.2). The cloud is the place where data and services are collected.
The service-oriented architecture pattern is most appropriate for this kind of SoS because the SoS offers services involving many constituent systems and requiring reconfiguration. In order to be resilient the system can only be loosely coupled.

### 3.2.3.2. Integrated Modular Avionics

**Description**

Integrated Modular Avionics (IMA) [Prisaznuk92] is a layered architecture pattern used in distributed real-time computer network airborne systems. IMA uses a layered architectural style, separating applications, the operating system and hardware. Each layer is modularised and connected by a network bus. Moving from a federated aircraft system to those employing an IMA architecture reduces cost, weight, size and power consumption, providing an improvement to the aircraft performance and maintainability.

Whether IMA systems may be considered SoS, is a contentious matter. IMA systems do meet many of the characteristics of typical SoS (they may evolve, have emergent behaviour, the component systems may be independently developed, are modularised and not interdependent. However, constituents are not geographically distributed and are centrally managed. As IMA systems share some properties of SoS, we shall consider them in this document, as “SoS thinking” would be valuable in addressing identified challenges.

Benefits include possibilities of dynamic reconfiguration, system evolution (though the addition of new components delivering new functionalities), the reuse of software modules, and a common/shared data stores and processing. Critical issues that arise in an IMA system include safety and modular certification (and recertification given reconfiguration), and performance.

**Industrial or theoretical case**

This is an industrial case describing a system that is already in operation.

**SoS Type**

As mentioned above, an IMA system is not necessarily an SoS, however it would most closely match a directed SoS.

**Architecture**

IMA systems implement the IMA pattern – that is a layered modularised architecture. The main layers are: application, operating system and hardware, shown in Figure 23.
These layers in Figure 23 allow the clear separation of the different software applications (labelled as Partition 1, 2 and 3) in an aircraft, whilst allowing a shared operating system and hardware – providing reusability of applications and reducing duplicate operating system code. Shared busses, processors and memory are present in the IMA system.

3.2.3.3. Road Traffic Management

Description
In the road traffic management SoS, Traffic Control Centres (TCC) are responsible for sections of the road network; collecting traffic information and providing traffic flow information to the public [D43.2 2013]. A wide collection of hardware sensors and actuators and software are involved, along with the human TCC operators. A TCC may govern a subsection of a road network, thus there may be several TCCs interacting across adjacent sections of the network – with contracts between them. Therefore, we may consider two SoS. One being an individually managed section of the network, including a TCC, and a collection of sensor and actuator systems; and also the next level up – being the collection of managed road networks.

Issues of importance include the control of each SoS: either a centralised control SoS, where decisions are made by the TCC, which enforces behaviour of the SoS
through the actuator systems; or through decentralised control where local subsets of sensor/actuator systems make local decisions based on their knowledge of the SoS. Another issue is that of integration of legacy systems into such a SoS, and the resulting need for SoS evolution.

**Industrial or theoretical case**
This is an industrial case describing a system that is already in operation.

**SoS Type**
The road traffic management SoS is an acknowledged SoS.

**Architecture**
The road traffic management SoS most closely matches the centralised architecture pattern (see section 4.3.1). The TTC system is considered the central system, with a collection of sensor and actuator systems. An evolution of the SoS may lead it towards a hybrid centralised/decentralised SoS, or even a fully decentralised SoS, however, this reflects only a hypothetical situation, and not the SoS currently in operation.

### 3.3. Relations to SoS Architectural Patterns

The following table relates the examples in this section with the architectural patterns presented in section 4.3 Describing Architectural Patterns.

<table>
<thead>
<tr>
<th>SoS Example</th>
<th>Pattern</th>
<th>SoS Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B&amp;O Case Study</td>
<td>Service Oriented</td>
<td>Collaborative</td>
</tr>
<tr>
<td>Insiel Case Study</td>
<td>Centralised</td>
<td>Acknowledged</td>
</tr>
<tr>
<td>London Emergency Services</td>
<td>N/A</td>
<td>Acknowledged</td>
</tr>
<tr>
<td>Health Care</td>
<td>Service Oriented or Publish-Subscribe</td>
<td>Directed / Collaborative</td>
</tr>
<tr>
<td>Finance and Investment</td>
<td>Service Oriented or Publish-Subscribe</td>
<td>Directed</td>
</tr>
<tr>
<td>Package Shipping and Tracking</td>
<td>Publish-Subscribe</td>
<td>Collaborative</td>
</tr>
<tr>
<td>Maritime Transportation</td>
<td>Service Oriented</td>
<td>Collaborative</td>
</tr>
<tr>
<td>Space Exploration</td>
<td>Centralised</td>
<td>Directed</td>
</tr>
<tr>
<td>Meteorology</td>
<td>Publish-Subscribe</td>
<td>Acknowledged</td>
</tr>
<tr>
<td>Air Traffic Control</td>
<td>-</td>
<td>Acknowledged</td>
</tr>
<tr>
<td>Defence Systems</td>
<td>Centralised</td>
<td>Directed</td>
</tr>
<tr>
<td>Reconfigurable Unmanned Aircraft</td>
<td>Centralised or Publish-Subscribe</td>
<td>Directed</td>
</tr>
<tr>
<td>Smart Grid</td>
<td>Service Oriented</td>
<td>Collaborative</td>
</tr>
<tr>
<td>Integrated Modular Avionics</td>
<td>Layered</td>
<td>Directed</td>
</tr>
<tr>
<td>Road Traffic Management</td>
<td>Centralised</td>
<td>Acknowledged</td>
</tr>
</tbody>
</table>

*Table 2: SoS Examples and respective patterns*
We can see that the service oriented, publish-subscribe and centralised patterns are present in the most examples (4-5 each). The popularity of the service oriented architecture (SoA) is a possible explanation for the present of this. The publish-subscribe pattern is also an obvious pattern for the future collaboration of constituent systems in SoS. Overall, there are only three (plus 1) different patterns among these examples which may point to a lack of diversity among the examples. On the other hand, there are two examples that deserve special attention. The absence of a pattern for the Air Traffic Control example may indicate that there may be more patterns to discover. As for the Integrated Modular Avionics, the Layered pattern is not initially considered a pattern for SoS. What this may suggest is that the Integrated Modular Avionics example is not a “true” SoS. This is an interesting idea since it may help us identify SoS by looking at the architectures used in their implementation.

Looking at the SoS types, we see that the examples are more or less evenly distributed between 3 types (4 Acknowledged, 6 Directed, 5 Collaborative). It is worth noting that there is no example of the virtual SoS type. Perhaps this indicates a need for further examples or perhaps the Virtual SoS type is simply difficult to apply in practice.

Finally, in term of trying to relate patterns and types, we see that, among our examples, the Service-Oriented pattern leads to SoS examples of either directed or collaborative type. The Publish-Subscribe pattern leads to SoS examples of either directed, acknowledged or collaborative type. The Centralised pattern leads to SoS examples of either directed or acknowledged type. However, the sample size is too small to make any definitive statements in terms of any overall relations between SoS types and patterns. Still, they might exist and that could be worth investigating.

3.4. Summary

Section 3 has presented and discussed characteristics for SoS. This was followed by a presentation of different either existing or hypothetical SoS in different application domains. The purpose of this was to try to recognize either existing or applicable architecture principles or patterns for system of systems. The identified architectural patterns are described in detail in section 4.3 together with other possible architectural patterns for system of systems. Finally the identified or possible architectures for the SoS application examples were related and shortly discussed in relation to both the architecture pattern and to the SoS type.
4. Patterns Collection

This section presents a collection of *modelling patterns* that can be used as part of a model-based approach to systems engineering. It begins by discussing the question of “what are modelling patterns?”, defines a standard approach to documenting modelling patterns that will be used in this document and finally presents a number of modelling patterns.

In this document two types of modelling pattern are considered:

3. **Architectural patterns** – Architectural patterns describe specific system architectures, both in terms of structure and behaviour, which address particular needs of the system. We consider the concepts of architectural patterns and architectural styles (introduced in Section 2.2.5) to be synonymous. An example of an architectural pattern would be one that addresses a particular type of control for a system (e.g. centralised vs. distributed).

4. **Enabling patterns** – Enabling patterns are specific constructs of modelling elements whose combination and subsequent use enables a number of systems engineering applications. An example of an enabling pattern would be one used for the definition of interfaces or one used to ensure traceability throughout a model of a system. While not used explicitly to define the structure and behaviour of an architecture, they are used to enable aspects of the modelling of an architecture, such as interfaces, evolution through time, testing, traceability etc.

Each of these types of pattern is discussed in a separate subsection below.

4.1. What are Modelling Patterns?

In 1977, Christopher Alexander, a professor of architecture at the University of California at Berkeley published an influential book on patterns in architecture [Alexander et al 1977]. In this book Alexander said that a "pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

Alexander's idea were adopted by elements of the software engineering object-oriented community in the 1990s before being made more widely known to software engineers through the publication of the so-called "Gang of Four" book in 1995 [Gamma et al 1995].

While initially aimed at the object-oriented software community, the idea of patterns spread to other aspects of software engineering with the publication of books on their use in such areas as analysis [Fowler 1997] and data modelling [Day 1996]. Since then the idea of a pattern - "an idea that has been useful in one practical context and will probably be useful in others" [Fowler 1997] - is being adopted into the wider systems engineering community. Indeed, many patterns
that were originally created for use in object-oriented software engineering are now being used in systems engineering modelling.

An example of this is shown in Figure 24 below. Figure 24, taken from [Gamma et al 1995], shows the composite pattern that is used “to represent part-whole hierarchies of objects”. It allows for hierarchies of any depth and width, consisting of composite objects made up of either other composite objects or terminal leaf objects. The example of usage of the pattern given by Gamma is concerned with the manipulation graphical objects in drawing editor software.

![Composite Pattern Diagram](image)

However, this pattern is equally applicable outside the software engineering domain. Figure 24 shows a typical system hierarchy nomenclature, with a system made up of subsystems that are made up of assemblies, which in turn are made up of subassemblies made up of components. Finally, components are made up of parts which cannot be subdivided further. The diagram has been annotated with stereotypes (the words in guillemots, « ») to show which element of the composite pattern each term is an example of.

The term modelling pattern is used in this document to simply refer to a pattern that can be applied to modelling aspects of a system, such as its architecture or its interfaces.

### 4.2. Defining Patterns

When defining modelling patterns it is useful to have a standard way of doing so. This helps give a consistent look and feel to each pattern definition, ensures that essential elements of the definition are not overlooked and aids in the understanding of the pattern.
We propose two main structures for describing patterns. The first, detailed in Section 4.2.1, is intended for the definition of architectural patterns. The second, in Section 4.2.2, is used to describe enabling patterns. Extensive use of both the UML and SysML is made in the definition of the patterns. See [Holt2004], [Rumbaugh et al 2005], [Holt&Perry2008] and [SysML 2012].

4.2.1. Describing Architectural Patterns

Given the design pattern outline in [Gamma et al 1995], we propose the following elements for describing architectural patterns:

1. **Background** – a description of the context and problem being addressed.
2. **Aims** – The aims (or intent) of the pattern.
3. **Ontology** – The concepts that the pattern addresses, together with the relationships between the concepts.
4. **Solution** – A description of the pattern structure – identifying key system elements (identified in the pattern concepts) and their relationships. The description should include a rationale to describe how the pattern addresses the identified problem.
5. **Example** – An example of the pattern in use with a SoS.

These elements are shown in Figure 26.
When documenting an architectural pattern the structure shown in Table 3 is adopted. This structure refers, where applicable, to the relevant pattern elements to be included in each section.

**Table 3 - Document structure for an architectural pattern**

4.2.2. Describing Enabling Patterns

Each enabling pattern is described using the COMPASS Architectural Framework Framework (CAFF) described in [D21.5b 2014]. As well as being used for the definition of full MBSE Architectural Frameworks, the CAFF is also suitable for the definition of enabling patterns.

Each enabling pattern thus:
1. Pattern Aims – The aims of the pattern, describing the purpose of the pattern. The aims are described using an Architectural Framework Context View (AFCV), one of the six CAFF Viewpoints.

2. Concepts – The concepts that the pattern addresses, together with the relationships between the concepts. Described using an Ontology Definition View (ODV), one of the six CAFF Viewpoints.

3. Pattern Viewpoints – Identifies the Viewpoints that make up the pattern and describes the relationships between the Viewpoints. Described using a Viewpoint Relationships View (VRV), one of the six CAFF Viewpoints.

4. Rules - A number of Rules are defined which ensure consistency between the Viewpoints and which define the minimum set of Viewpoints that are needed when using the pattern. Described using a Rules Definition View (RDV), one of the six CAFF Viewpoints.

5. Viewpoint X – A description of the Viewpoint in terms of the Ontology Elements from the pattern’s Ontology (found on its ODV) which can appear on the Viewpoint. Each Viewpoint is intended to represent a particular aspect of the pattern, but Viewpoints can overlap in the information they present i.e. concepts from the Ontology can appear on more than one Viewpoint. The Viewpoints must cover all the concepts from the Ontology i.e. there must be no concept from the Ontology that does not appear on at least one Viewpoint. Each Viewpoint has its aims described using a Viewpoint Context View, one of the six CAFF Viewpoints. It’s content is then described using a Viewpoint Definition View, another of the six CAFF Viewpoints. Example Views that conform to the Viewpoint are then given. This is repeated for each Viewpoint in the pattern.

6. Extensions – (Optional section). Any possible extensions to the pattern are described.

7. Summary – A summary of the pattern.

This structure is used for each of the enabling patterns found in Section 0.
4.3. Architectural Patterns

Architectural patterns describe specific system architectures, both in terms of structural organization and behaviour, which address particular needs of the system. We consider the concepts of architectural patterns and architectural styles (introduced in Section 2.2.5) to be synonymous. An example of an architectural pattern would be one that addresses a particular type of control for a system (for example, centralised control), or the use of shared resources in the SoS.

The use of architecture patterns tends to be limited to software engineering, though their use has been proposed in systems engineering [Cloutier&07]. To date the use of architecture patterns in systems engineering is at its infancy, and in SoS architectural patterns is considered an important challenge [TAS&12]. Patterns tend to appear through the repeated development of software/systems/SoSs. We focus both on existing patterns from literature – and consider how they may be applied in a SoS, where the basic architectural elements of the patterns are the constituent systems of an SoS – and develop several new architectural patterns.

We begin to consider those architectural patterns, which may be used at the SoS-level. For example, in this work the model-view-controller pattern [Reenskaug79] and the patterns described by Gamma et al. [Gamma et al 1995] consider paradigms relevant only to software and do not translate to SoS. As such, in this deliverable, we consider a set of patterns to include those identified in our survey of SoS examples in Section 3.2:

- Centralised
- Service Oriented
- Publish-Subscribe
- Pipes and Filters
- Supply Chain
- Blackboard
- Infrastructure Grid
- Reconfigurable Control Architecture

When considering such architectural patterns, we also consider how the patterns relate to the architecture principles and SoS types outlined in Section 2 and 3.

Finally, it’s not our expectation that an SoS will exhibit one pattern only. Most SoSs are capable of implementing multiple patterns at once. For example, a Publish-Subscribe pattern may be employed along with a Reconfiguration Control Architecture.
4.3.1. Centralised Architecture Pattern

4.3.1.1. Introduction

*Background*
A centralised architecture (similar to the star network pattern) is an architecture pattern whereby there is some central point of control of a SoS. The central constituent system is connected to the other systems of the SoS and is responsible for ensuring the correct behaviour of the SoS. An example centralised SoS, given in more detail later in this section, is a military command and control SoS, where a central system controls the constituent systems to achieve the aims of the SoS. There may be degrees of centralisation in an SoS, for example a fully centralised SoS connects all constituents to a single central system. Alternatively, we may consider a central system that is connected to a series of sub-central systems. A centralised architecture may appear to deny autonomy of the constituents, because the SoS emergent functionality is dependent on a single centralised constituent. However, constituents are till capable of exhibiting some autonomy in a centralised SoS. The central constituent, for example, has the ability to take decisions about functionality it delivers and will deliver in the future, and to replace, add or remove constituents upon which it depends. The constituents surrounding it may not even be aware of participating in an SoS (e.g., they may be commercial off-the-shelf systems) and their ability to make autonomous decisions (e.g., to change their services and functionality) can continue unabated. Of course, certain changes may result in their removal from or replacement in the SoS.

*Aims*
The main aims of this pattern are to support:

- Centralised control and management of SoS
- Reuse of pre-existing systems

*Concepts*
The concepts of the centralised architecture pattern are outlined in Figure 27 below. A centralised SoS is composed of a single central system, and several constituent systems. The central system is connected to all constituent systems and controls the SoS.
4.3.1.2. Solution

Overview
A centralised architecture is, perhaps, the simplest of SoSs to engineer from scratch and is closest to the directed and acknowledged SoS types as described by Dahmann [Dahmann & 2008]. The central system aims to use functionality of constituent systems, and directs and controls the constituents of the SoS. There are degrees by which a SoS may be centralised. From a completely centralised SoS – where all constituent systems are connected to the central system – to a partly centralised SoS, where a central system is connected to constituents which in turn are connected to other constituent systems. This is discussed in more detail later in this section.

Structure
A SoS may be considered as centralised when a single constituent system is the driving and managing force of the goals and functionality of the SoS. As such, the central system may be considered the ‘owner’ of the SoS.

As stated in Section 3, one aspect of differentiating between the classifications of SoS is the extent to which the constituent systems are designed to function as a part of the SoS. In a centralized SoS, the central system is typically developed explicitly to achieve the SoS goals and functionality. As such, a centralized SoS should be thought as a directed or acknowledged SoS. The other constituent systems of the SoS provide functionality leveraged by the central system so to deliver the functions of the SoS. These systems may be legacy or pre-existing.
It is the responsibility of the central system to ensure compatibility with the other constituent systems for the good of the SoS.

An important distinction to make when visually inspecting a topological architectural diagram of a SoS, is between a constituent system which is used by several other constituent systems to provide a service, and a centralised architecture. In the former case, this ‘common’ system provides some service to the other systems, however enacts no (or little) control over the SoS and does not aim to address the goals of the SoS. When defining a centralised SoS architecture, it is important to ensure the description makes clear that the central system provides this control. This should be described in behavioural models of the SoS. In SysML, for example, we may consider using the behavioural diagrams alongside structural diagrams indicating the central system requires interfaces provided by the other constituents.

The extent of centralization may differ between SoS; we consider a fully centralised SoS and a hybrid centralised-distributed SoS. In a fully centralised SoS, a single constituent system is responsible for delivering the SoS goals and functionality and connects directly to a collection other constituent systems in order to achieve them. In Figure 28 below, the constituent system S1 may be considered the central system, and is connected to the constituents CS1, ..., CS4.

In a SoS which has a single central constituent, we may consider a degree of decentralization of the connecting constituents. For example, we may consider a central system requires some defined functionality and connects to a different constituent system which is itself considered a central system in a sub-SoS,
controlling/managing another collection of constituents. For example, consider the SoS in Figure 29 below, in which the central system S1 is connected to the constituents CS3 and CS4, which are central systems using constituents CS5 and CS6, and CS 7 and CS8 respectively.

![Figure 29 - SoS with hybrid centralised-decentralised architecture](image)

In a *hybrid centralised-distributed SoS*, the centralization of the SoS may be distributed over a core of several constituents. Distributing this centralised control allows a SoS to benefit from the explicit management of the SoS, but lessens the reliance placed on a single central system. The degree to which a SoS may be have distributed control and still be considered centralised is subtle, and beyond our ability to reason at this time.

It may be noted that the Service Oriented Architecture pattern described in Section 4.3.2 may be a specialization of a centralised SoS. In this case, a system providing the underlying composition and orchestration of services may be considered the central constituent system.

**Rationale**

In this subsection, we consider how the solution structure addresses the aims of the centralised architecture pattern stated in Section 4.3.1.1.

The first aim, centralised control and management of SoS, is achieved through the use of a single constituent system (or in a hybrid-centralised SoS, a group of systems) to be responsible for achieving the goals and functionality of the SoS. Having a single centralised system controlling and managing the SoS allows an SoS to be designed and thus verification is possible in the early stages of SoS design.
Whilst the central system of the SoS would typically be bespoke to that SoS, as it is designed for the use by the SoS, it is possible for the other constituent systems to be existing systems – the second aim of the pattern.

**Example**

We may consider several SoS which exhibit the use of the centralised architecture pattern, these typically fall into those SoS with a command and control-type purpose. SoS is a strong field in the military domain, and this domain provides a strong example of a SoS which applies a centralised architectural pattern. The anti-guerrilla operations SoS example, depicted in [Hall-May&06] and illustrated in Figure 30, has a central Theatre Command system, a system comprised of UAV scouts, and constituent systems including artillery, troops and the required communication infrastructures. In the example SoS, the theatre command system (which includes a strong human aspect) makes operational decisions based upon data sourced from UAV scouts and other sources, to give commands to the various troops and artillery. The goals of the SoS – in this case to suppress enemies – are achieved due to the commands of the central constituent system. It is the responsibility of the central command to ensure the operation SoS functions are provided.

An alternative example is in the Traffic Management SoS, depicted in Figure 31 below. In this SoS, the central constituent system is the Traffic Management System (TMS), which primarily controls the Road-Side Actuator System (RSAS), given information obtained from the Congestion Monitoring System (CMS). The TMS includes elements including human operators, computing systems, radio and telecoms equipment. The CMS is primarily a hardware and software system with a large number of distributed simple sensors, however the CMS may also include humans providing the runtime traffic data. The RSAS is again a hardware and software system, which may include a large number of legacy systems due to the size of such a managed road network.
The TMS obtains information from the CMS, including traffic speed, through the Mon_IF. In the example, the CMS is polled for traffic data through the Mon_IF interface, though this data could also be pushed through an interface provided by the TMS. The TMS sends commands to the RSAS through the Act_IF interface provided by the RSAS. These commands include the ability to set speed limits, divert traffic, open/close road access and road side messages to road users. This element of the system uses a classical control-loop type system.

The SoS may also control elements of the Highways Agency (HA) system – for example for deploying more road-side actuators (including humans), and also for providing commands and information for further maintenance on the road network. The final element of control in this SoS occurs over the TP_IF with the Transport Police (TP) system. This control allows the Traffic Management System to command the TP to police a section of the road system – for example an accident at the source of congestion.

There is scope for this system to have some element of decentralised command – whereby the HA and TP systems may override or influence decisions the TMS makes. This may occur, for example, when the HA wishes to close roads for construction, or the TP wishes to divert traffic for security reasons. For this reason, the use of the centralised pattern may be tailored for the SoS.

4.3.1.3. Summary

The centralised architectural pattern provides a central constituent system responsible for the control of an SoS. The central system is typically designed to
take advantage of functionality of the other constituent systems of the SoS, so to ensure the SoS achieves its goals.

A level of distribution may be afforded to remove the reliance on a single constituent and to distribute the control, whilst maintaining the overall aims of the pattern.
4.3.2. Service Oriented Architecture

4.3.2.1. Introduction

Background
Service Oriented Architecture (SOA) is an architecture pattern developed for software applications whereby applications are constructed through the use of third-party services. Services are stateless – that is they have internal state, but do not share state – to the application, they act as functions on supplied data. Using the SOA pattern, a designer may construct applications by selecting services by their offered service description. Service providers may develop systems in any way they wish, such that they provide standardised descriptions and expose a means for provision of the service.

Aims
The main aims of this pattern are to support:
- Analysis of SoS emergent behaviour
- SoS/constituent system evolution
- Central SoS authority
- Enable cross-domain SoS development
- Long SoS lifecycle

Concepts
The concepts of SOA are outlined in Figure 32 below.

Figure 32 - Service Oriented Architecture Architectural Pattern Ontology
A service provider is an organisation or constituent system, which may provide several services. Each service may be provided by several different providers. Services are accessed through service interfaces that contain functions. The interface exposes a subset of the functions of the service and conforms to a specified protocol, subject to a security policy.

Services publish a service contract constituting a service description and a service-level agreement. The service description provides details of the service interface – in particular the offered functionality. The service-level agreement details a set of quality of service (QoS) guarantees – to which the service interface functions are subject to. The service description is used as a basis for the discovery of services.

A service client uses a service, through the service interface subject to the service-level agreement.

4.3.2.2. Solution

Overview
The SOA architectural pattern has many implementations in software architecture, although is primarily associated with Web Services [WS04, Palpazoglou08] whereby services are described using WDSL [WSDL01], and interactions between service providers and service requesters communicated using SOAP [SOAP07]. In this section, we consider how the concepts of the pattern may apply to SoS, and what underlying architectural elements are required in a SoS to enable SOA concepts including discoverability, contract agreement and composition. There are several areas of research in the application of SOA to SoS, including the IMC-AESOP project (http://www.imc-aesop.eu).

Structure
As described in Section 2.1.1, a SoS is composed of several constituent systems. The constituent systems expose their interfaces, which act as points of interactions between constituents. In applying SOA to SoS, we propose a similar relationship between services in SOA and components in component-based systems.

The relationship between SOA and component-based software architectures is subtle, and often confused. In software architectures, a software system is composed of several distinct components connected in some configuration. In SOA, services are an abstract means to consider the functionality provided by a system. How the system providing a given service is defined is the concern for software architects. For example, in Figure 33 below, a system composed of components A, B and C, composed into a configuration, provide two different services – mapping and routing. The components are connected through component interfaces, which may in turn be used as service interfaces.
Applying SOA to the SoS level, we propose that collections of constituent systems may be combined to provide services.

When designing a SoS with the SOA pattern, we would require two separate notions of an interface: a constituent system interface (CS-Interface) and a service interface (Serv-Interface). The design process should first consider only services – that is what services are required to provide the functionality the SoS must provide to its environment – and not the individual constituent systems. A SoS, designed with the SOA pattern therefore is centralized in nature – and in the terms of SoS classification would be considered directed or acknowledged. This is due to the fact that there must be some entity involved with the selection of services, however this entity has no power over the management or operation of the constituents providing the services (indeed, if taking a purely service-oriented approach, they may not be aware of the underlying constituent systems).

Serv-Interfaces define the functionality provided by services and may conceivably be defined in a similar manner to normal CS-Interfaces. When we model the constituent systems implementing a service, the relationship between the CS-Interfaces and the Serv-Interfaces is important, and requires further consideration. As such, we propose the use of the Interface enabling pattern, defined in Section 4.4.1, to define Serv-Interfaces.

Each service must also define a service contract. The contract should include a description of the service functionality i.e. it should reflect the Serv-Interface definition. The contract should also define a service-level agreement defining QoS guarantees made regarding the functionality specified by the Serv-Interface. It is possible, therefore, for a service to have several service contracts for the same Serv-Interface. The contracts may differ by their service-level agreements. For example, a mapping service may have two contracts: one for civilian use and

---

3 Initial considerations include; a one-to-one mapping – where a Serv-Interface is realised as a single CS-Interface, or a one-to-many mapping – where the responsibilities of a Serv-Interface are split between collections of CS-Interfaces.
another for industry. The contracts may describe the same functionality, but vary by providing different image resolutions or response times.

The next stage of designing a SOA-SoS is to define the underlying service-oriented mechanisms required to handle services. The extent to which these elements are required will differ between SoS. These elements constitute:

- Service Contract Publishing
- Service Discovery
- Service Binding
- Service Composition and Orchestration
- Contract agreement

It is beyond the scope of this deliverable to state how these elements should be implemented in an SoS.

**Rationale**

In this subsection, we consider how the solution structure addresses the aims of the SOA pattern stated in Section 4.3.2.1.

As a type of centralised pattern, the SOA pattern provides a central system, which composes services to achieve the SoS goals and functionality. This satisfies the aim of central SoS authority. SoS evolution is enabled through the loose coupling of SoS management. Service providers need not know how a service is being used – only that they must meet the guarantees made in the service contract. This allows the SoS to evolve through dynamic binding of services.

Through the separation of the service contract (and in particular the service interface) and the underlying service implementation, the pattern achieves the aims of constituent system evolution and cross-domain SoS development. A service consumer may use a service without requiring knowledge on the service logic or implementation. From a service provider’s perspective, this enables evolution of the architecture of the constituent systems (or the collection of constituent systems) providing the service without requiring a change of service contract (assuming the contract still holds) and also allows the constituent system to be developed in any way, such that a service interface is defined.

The analysis of SoS emergent behaviour may be achieved through the analysis of the service descriptions – both the functional and non-functional aspects. However this requires further effort in understanding the relationships between system and service interfaces, as described earlier.

Finally, the SOA pattern supports a long SoS lifecycle through the use explicit separation of service interface and service implementation. This enables constituent system developers to apply different methodologies and development processes.

**Example**

The SoS application examples in Section 3.2 present several examples of the use of SOA. Here, we describe a travel agent booking management SoS example that may be defined using the SOA architectural pattern.
In the SoS, described with a SysML internal block diagram in Figure 34, a central front-end system receives requests from its environment (either a consumer or travel agent) to book a trip consisting of a hotel, airline (and possibly other aspects of a holiday/business trip including car hire, insurance which are not shown here). The front-end system is responsible for receiving trip requests and responding with trip details. The front-end, therefore is required to use services provided by several service providers to create a trip booking, and thus the front-end system is responsible for providing the functionality of the SoS. The functionality provided by the front-end may also include compensation for cancelled/over-booked trips and suggesting trip dates to ensure all service may be provided to the user.

A travel service discovery system is also present in the SoS which is responsible for retrieving service contracts from the different service providers and ensuring that the front end system uses the services most suited to the SoS. This may conceivably be an automated process, but it may also be a highly complex system requiring legal entities and contract negotiation. Figure 34 depicts two services: the *Hotel Provider Service 2* and *Airline Provider Service 3*, as part of the SoS environment, providing service contracts to the discovery agent. Whilst the diagram in Figure 34 does not depict dynamic reconfiguration of the SoS, the front end may change service providers dynamically – for example using the *Hotel Provider Service 2* service rather than the *Hotel Provider Service 1* service. The loose coupling of the SOA pattern ensures that the SoS is amenable to such reconfigurations, however issues surrounding reconfiguration are still present – such as quiescence (whereby the systems should be in a state such that
transactions are not currently taking place). This is an issue beyond the scope of this deliverable, and not restricted to the SOA pattern.

As mentioned earlier, component-based systems using the SOA pattern and an SoS using the pattern have a similar relationships when considering systems and services. In Figure 35 below, the three services used in the travel agent SoS are depicted with the constituent systems of the SoS along with the interfaces between those constituents. The dashed line in the figure identifies the service boundaries. In Figure 35 it is clear that there are shared systems between services, for example the Syndicated Travel Agent Group System is a part of the Hotel Booking Service 1 and Airline Provider Service 1 services.

The Serv-Interfaces identified in Figure 34; H1_Serv_IF, A1_Serv_IF and A2_Serv_IF are provided solely by the Syndicated Travel Agent Group System and Airline Booking System constituent systems.

![Figure 35 - Constituent system composition of travel agent services](image)

Whilst in this example (and in Figure 35 above in particular) the internal composition of the constituent systems of a service is outlined, the SOA pattern shares with a characteristic of SoS that the services are encapsulated – that is the front-end system knows only about a service’s interface and contract. The front-end (and thus the SoS engineer) cannot be aware of the internal composition of a service, unless the services are owned and managed by same organisation as the front-end. This reflects the characteristic of managerial independence of SoS as highlighted by Maier [Maier98].

This example emphasises the fact that a SoS using this design pattern cannot be considered a virtual SoS as defined by Dahmann [Dahmann&2008]. The presence of a discovery system and front-end system performing the choreography (so to
produce the SoS-level functionality) shows that the SoS is designed and controlled by a (collection of) constituent systems.

4.3.2.3. Summary

The SOA-SoS pattern, derived from the SOA software architectural pattern, provides reusable services which are loosely coupled, both between services and the management of services. The externally visible functionality of services is defined by their interfaces, which are described by service contracts – used as the basis of service discovery and composition.

Using a service-oriented pattern, abstracts from the SoS architecture in terms of constituent systems – how services are provided is not of concern. Services are reusable between SoS, and the loose coupling ensures they are composable and by specifying service interfaces, promotes interoperability.
4.3.3. Publish-Subscribe Architecture Pattern

4.3.3.1. Introduction

Background
Publish-Subscribe is an architecture pattern and a communication paradigm developed for dissemination of information between distributed software centric systems.

This pattern can be divided in two different subgroups: an Event-Based Publish-Subscribe pattern (EBPS) and a Data-Centric Publish-Subscribe (DCPS) pattern. This description will shortly introduce the event-based publish-subscribe version but focus on the data-centric publish-subscribe version as it is more useful as an architecture pattern for SoS.

The Publish-Subscribe pattern has been widely used in industrial systems and recently the DCPS paradigm has been standardized by OMG as the “Data Distribution Service for Real-Time Systems” (DDS) standard [OMG-DDS2007]. DDS is a standard for a data-centric architecture for dissemination of information between heterogeneous distributed systems and currently supported by a number of commercial middleware implementations as well as open source implementations. DDS specifies a Data-Centric Publish-Subscribe (DCPS) model, where DCPS provides the functionality required for an application to publish and subscribe to the values of data objects of given types.

Aims
The main aims of this pattern are to support:

- **Loose coupling** between publisher and subscriber constituent systems in the SoS with **time, flow and space decoupling** between the constituent systems
- **One to many and many to many communications** between the constituent systems in the SoS

Concepts

Concepts in the Event-based Publish-Subscribe pattern
In the EBPS pattern events are exchanged between publishers and subscribers based on service oriented publish and subscribe interfaces. The concepts of the EBPS architecture pattern as described in [POSA4-2007] are outlined in Figure 36 below.

The **Publisher** is the entity responsible for publishing an event via the **ChangePropagationInfrastructure** to the associated subscribers who have registered for the actual event type.

A **Subscriber** is responsible for receiving the published events from the **ChangePropagationInfrastructure**.
The *ChangePropagationInfrastructure*, also by other sources called an event channel, is responsible for handling the registration of subscribers and for the actual dissemination of a given event to the set of registered subscribers.

**Concepts in the Data-Centric Publish-Subscribe pattern**

Figure 37 introduces a simplified conceptual model for the Data-Centric Publish-Subscribe (DCPS) pattern and Figure 38 shows the complete DCPS model.

A *Topic* describes a *Data-Object* with a unique name in the given domain, a data-type and a set of Quality of Services (QoS) related to the data.

A *Publisher* is the entity responsible for data distribution to a set of registered subscribers on the network and publishes data on one or more topics.

A *DataWriter* is the entity used by the publishing application to publish data of a given type associated with a unique Topic. The *DataWriter* use the *Publisher* for the actual distribution of the new value.
A **Subscriber** is the entity responsible for receiving published data on one or more topics and making these available to the application at the subscriber site, which has subscribed for the data associated with one or more topics.

A **DataReader** is the entity the application uses to access the received data from the attached subscriber.

Both a **DataWriter** and a **DataReader** have typed interfaces for a given topic acting as a mediator on the publishing site to the publisher and on the receiver site to the subscriber.

The additional main concepts are:

A **DomainEntity** is the super class for **Publisher**, **Subscriber**, **Topic**, **DataWriter** and **DataReader**, where each of these can have an associated set of **QoS Policy**, have an optional **Listener** or an optional **StatusCondition** associated. All of these domain entities are associated with a given **DomainParticipant**. **Listeners** and **StatusConditions** entities are used as communication mechanisms between the application and the DCPS middleware.
A DomainParticipant represents a given domain, where the attached publishers and subscribers can communicate and exchange information by publishing and subscribing on named topics defined in the given domain.

A QoS Policy describes a given quality of service element and can be attached to all entities in the model. The DDS standard specifies currently 22 different QoS parameters [OMG-DDS 2007]. We also consider QoS aspects with respect to interface definition in Section 4.4.1.10.

4.3.3.2. Solution

Overview
In this section, we consider how the concepts of the Data-Centric Publish-Subscribe Architecture pattern may apply to create a SoS based on a set of constituent systems.

Structure
In a SoS each constituent system can play the roles of one or more publishers, one or more subscribers or acting as both publishers and subscribers to on one or more Topics.

A given set of CS can participate in a specified communication domain (DomainParticipant). Some of these CSs can be connected to other communication domains as well. A given SoS can in this way be defined as one or more communication domains each with its own set of CSs as indicated on Figure 39, where CS1-CS5 participates in domain 1 and C4-C8 in domain 2. In this example both CS4 and CS5 participates in both domains, which can be realized as a gateway between the two domains if needed as indicated in CS5 or only participating in the two domains without interactions between the domains as indicated in CS4.

Figure 39 - Example of a SoS with two domains
This architecture pattern will be very useful in building the collaborative types of a SoS, where each CS agrees on a common data model consisting of a set of topics to be used in the collaboration to exchange information between the CS. In this architecture pattern there is no central system or no CS acting as a central node. To obtain a dependable system a given publisher CS can be made redundant by allowing a backup CS to publish on the same topic and taking over a hot stand by backup CS in case the primary CS fails.

If a CS acting as a subscriber wants to join or leave a given communication domain it can do this simply by registering or de-registering on a given Topic defined in the actual domain.

**Rationale**
In this subsection, we consider how the solution structure addresses the aims of the Data-Centric Publish-Subscribe pattern stated in section 4.3.3.1.

The first aim, loose coupling can be more specific described by time, flow and space decoupling. Time, flow and space decoupling, is achieved by the nature of the Publish-Subscribe paradigm. Time decoupling means that Publisher and Subscribers do not need to be online at the same time for sending or receiving data as the middleware system has the possibility to store data for a specified period of time. Flow decoupling means that a publisher can publish its data asynchronously without waiting for a subscriber to receive it. On the subscriber site a subscriber do not need to wait on a given data but can be notified asynchronously when a change in subscribed data occurs. Space decoupling means that either the publisher or the subscriber needs to know the identity of each other.

The second aim, one to many and many to many communication, is achieved by the nature of the Publish-Subscribe paradigm and is also specified by the DDS standard. There can be one or more publishers on the same topic and one to many subscribers on the same topic. The case with more publishers can be used to obtain redundancy on the publication site.

**Examples**
As an example, consider the SoS shown in Figure 40 (see the Health Care Systems in section 3.2.2.1):
The central concept in this example is the Medicine Card topic, which contains the actual medicine prescriptions for a given citizen along with the history of prescriptions. Several constituent systems can update this information as publishers to the Medicine Card information, represented as a topic with a number of attributes. Several other constituent systems subscribe to the medicine card information and receive updates when the medicine prescription changes for a specific citizen which is subscribed for by one of the other systems shown. The underlying interaction mechanism is a push-mechanism, where changed information is pushed to all the registered subscribers. This pattern has a very loose coupling between publishers and subscribers, where it is very simple to add new publishers as well as new subscribers.

As a second example consider the SoS shown Figure 41, which is based on the Package Shipping and Tacking system presented in Section 3.2.2.3.
In this example the different package shipping systems publish tracking information to the Package topic. Several systems can subscribe to this information and be updated whenever the data for a given package subscribed for change. With this pattern it is very easy to add a new shipping system as a publisher and in the same way it is easy to add more subscribers. The Package Tracking CS-System could also have a service-oriented interface for other constituent systems which could query a package status through such an interface.

4.3.3.3. Summary

The Data-Centric Publish Subscribe architecture pattern originally developed for distributed software oriented systems looks as a promising candidate to be used in development of SoS due to its high degree of decoupling between the constituent systems in an SoS. It can be used to support the collaborative SoS system type, but will also be applicable to both directed and acknowledged systems.
4.3.4. Pipes and Filters Architecture Pattern

4.3.4.1. Introduction

Background
The pipes and filters architecture patterns is originally developed for data flow oriented software systems but can be used more generally for systems of systems, where the flow can be material, liquid or energy flow.

It is described in [Shaw&Garlan 1996] as an architecture style and in [POSA1 1996] as an architectural pattern.

Aims
The main aims of this pattern are to support:
- Data or material flow oriented systems
- Independent processing steps on a flow
- Configurable transmission of the flow between processing elements
- Dynamic change of processing steps and connectors

Concepts
The concepts of the pipes and filters architecture pattern are outlined in Figure 42 below.

The Filters in this pattern represents the processing steps, where e.g. data or materials are processed from one input form to an output form.

The Pipes in this pattern represents the pipeline connecting the Filters for transferring the data or material from one processing filter to the next and the pipeline connecting the input source with the first filter and the pipeline connecting the last filter with the output sink.

The Input Source represents the first step where the data or material enters into the system through the input source.
The **Output Sink** represents the final step where the resulting data or material leaves the system through the output sink.

Important invariants are that the filters must be independent and not share state with other filters and do not know the identity of other filters in the processing line.

### 4.3.4.2. Solution

**Overview**

In this section, we consider how the concepts of the pattern may apply to SoS, and what underlying architectural elements are required in a SoS are required for using this pattern in a SoS context.

**Structure**

This pattern can be applied for either directed or acknowledged system of systems, where there is a common and recognized goal for the SoS. It could in principle also be used in collaborative SoS, where a given CS voluntarily joins a specific place in a processing chain of other CS and in this way gives an extra service, which was not possible otherwise.

In a SoS based on this pattern each constituent system can be either a system working as a filter-CS doing processing or transformation of the flow or a CS can be a pipe-CS doing only transport of the flow or possibly also an intermediary storage. Finally a CS can act as an input-source-CS or an output-sink-CS.

The complete SoS is described by the set of CS connected as: input-source-CS, \{pipe-CS, filter-CS\}*, pipe-CS, output-source-CS. Where the \{pipe-CS, filter-CS\} sets can be dynamically changed during the system lifetime. This is described in Figure 43.

![Figure 43 - SoS Pipes and Filter pattern](image)

A modified pipes and filters system allows configuration of filters and/or as an alternative addition of materials to the processing steps. This was inspired by the Modified Pipes and Filters Model [Shaw&Garlan 1996], as shown in Figure 44. The SoS-Control & Configuration-CS can be used to implement either a directed or acknowledged SoS, where this central CS can enforce global control over how the
Filter-CS processing is performed e.g. start or stop or just configuring the processing steps performed by one or more of the independent Filter-CS.

Figure 44 - Modified SoS Pipes and Filter Pattern with control & configuration

Another modification to the basic Pipes and Filter pattern is shown on Figure 45, where a given filter is allowed to have either more than one input pipes or on the output site with more than one output pipe. This pattern modification could be combined with the control & configuration possibility as well, where the control could decide upon the number of input channels as well as the number of output channels to use for a given filter.

Figure 45 – Modified SoS Pipes and Filter Pattern with more inputs or outputs

**Rationale**

In this subsection, we consider how the solution structure addresses the aims of the pipes and filters pattern stated in section 4.3.4.1.

Support for *data or material flow oriented systems* is realized by the serial connections of the pipe and filter components, where data or material flows through the pipes to a filter or processing component and where the output from a given filter component is send further down the processing line.

The aim of *independent processing steps on a flow* is realized by having independent filter processing components along the processing line, which can be implemented as an independent CS.

The aim of *configurable transmission of the flow between processing elements* is realized by the pipe component, which can be of different type and in principle being a CS in its own.
The aim of **supporting dynamic change of processing steps and connectors** is realized by the structure where both filters and pipes can be exchanged during configuration or in some cases also be dynamic configurable during run-time.

**Examples**

As examples, consider the two SoS examples below.

**Software oriented SoS:**

This SoS example consists of seven different constituent systems each with its own purpose but together gives a new and enhanced functionality. The first *LocalMonitoring-CS1* system monitors a patient’s vital signs in a local setting e.g. in a private home, with the possibility to make local storing and display of monitoring results and give local alarms. The signals monitored can be forwarded through the *Pipe-CS2* system to a *CentralMonitoring-C3* system, where further analysis of the data can be performed by e.g. monitoring by a doctor. This system can receive input streams from many local systems. The processed signals can be forwarded to *Pipe-CS4* for centralized storage of the signals for the given patient. This *Central-Storage-CS5* is a system of its own which can have many users that use and requests the stored data.

In this example the three *Pipe-CSs* can be thought of as a kind of independent transport-systems which for example could implement a secure and safe data transfer service and delivery between two endpoints, in this case between CS. In some situations where data is transferred the Pipe-CSs would be implemented by standard middleware software and in this case they will disappear as independent CS.

This architecture gives a fine decoupling of the CS as the receivers of a given flow can be redirected dynamically to another filter component.
Material oriented SoS:

In this example the SoS as a whole is assembling complex items e.g. parts to more complex systems like cars or aircrafts, where each Assembly-CS is considered as a stand-alone CS e.g. it could be an independent factory with a specialized capability for either assembly or processing of complex items. In this system the Pipe-CS is the transportation systems, which again could be independent companies e.g. shipping companies. The SoS-Control-CS represents the CS which manufactures and sells the complex end-products. The SoS-Control-CS interacts with the Assembly-CS (the subcontractors) to order a specific part product for the next assembly step.

4.3.4.3. Summary

The pipes and filter architecture pattern is a candidate architecture for SoS where the constituent systems are connected in a serial manner and working on a common stream of data or material, where each CS transforms the input stream to an output stream and where the CS can be either a filter-CS or a pipe-CS.
4.3.5. Supply Chain Pattern

4.3.5.1. Introduction

**Background**
The supply chain pattern has some similarities to the pipe and filter pattern, but also some important differences, and so we consider it separately.

In an SoS that implements the supply chain pattern, the overall SoS goal is the ultimate delivery or manufacture of a final product or service. Some CSs will fulfil the role of suppliers, and some CSs play the role of integrators who assemble supplied components into further, more complex components or final products. Some CSs may be integrators and suppliers.

A subset of CSs are logistics suppliers, providing transportation services to move goods between suppliers and integrators. This function is frequently outsourced and handled by third party logistics specialists. CSs are often widely geographically distributed in this type of SoS, perhaps across multiple continents, so logistics and sharing of status updates are important. The transportation or logistics CSs mirror the role of the “pipe” in a pipe and filter pattern, in that a flow of supplies is passed to the logistics CS and a flow is produced by them to the next CS in the chain. However, logistics CSs in a supply chain typically supply detailed status information (e.g., current location of shipments, estimated arrival and any foreseen delays) which a pipe would be not be expected to produce.

A significant CS in the chain is the final integrator, responsible for delivery of the final product. The final integrator may have substantial influence over the rest of the chain, but there is usually no overall director over the SoS.

A flow of goods moves through the chain, from the first supplier towards the final integrator. Each CS typically shares a contract with their neighbours that stipulates the services/goods they will deliver, quality of service and timings. An example of supplier, integrator and logistics CSs and their interfaces is shown in Figure 48; in this figure the supplier and integrator share interfaces for negotiating contracts; and the supplier shares interfaces with the logistics CS for negotiating a contracts. The logistics CS has an interface for making status updates available to both CSs, and the supplier makes status updates available to the integrator.
This example represents a small SoS; many supply chain SoSs may include many suppliers, logistics suppliers, and integrators.

We characterise two different sub-types of the supply chain pattern below:

- **Closed supply chain.** In a closed supply chain, adding new CSs or removing existing ones is difficult. This is because there is little standardisation in the way of processes, data, and products. As a result, each CS produces bespoke products and/or shares data about their processes and current status in a non-standard way; it is time-consuming to replace a CS in this chain, because a replacement must be sought and negotiated with before they may join the chain. In return, CSs may find it difficult to depart the chain, since their products and processes are heavily invested in the current tailored supply chain and may not be easily adapted to a new SoS.

- **An open supply chain.** In an open chain, adding new CSs, or replacing existing CSs, is more straightforward, because processes, data, and products are standardised. As a result, CSs are more mobile, and can join the SoS, depart it or be replaced more quickly and easily.

There may be multiple levels of integrators and/or suppliers. Any of the integrators or suppliers may be participating in multiple supply chains – e.g., suppliers may also provide inputs for other, perhaps competing, supply chains. Suppliers may be obliged to withdraw from the supply chain if they develop an exclusive contract to supply a competitor.

The supply chain pattern and its variations differ from the pure pipe and filter pattern in the following ways:
- the CSs in a supply chain are often aware of the ultimate SOS goal, particularly in a closed chain.
- Transportation suppliers ("pipes") are active participants and share data about current status with the CSs in the chain.
- Transportation suppliers also typically provide their services to other parties unconnected with the current chain.
- CSs may be aware of the internal state of their peer CSs.
- CSs are capable of generating input and feedback for CSs upstream – for example, returning supplies which do not meet contractual obligations. In this way information about events and supplies upstream can allow CSs further downstream to take action early to avoid the possibility that production is affected adversely by delayed supplies.

**Aims**
The main aims of this pattern are to support:
- Efficient and fast product manufacture and delivery across large distances
- Collaboration between suppliers and integrators
- Easily configurable chains, allowing chains to adapt to new opportunities or to problems that arise

**Concepts**
A supplier is a CS that provides an input for other CSs which consume them. An integrator is a CS that accepts inputs from multiple CSs and produces an integrated output. CSs may play the role of integrator, consumer and/or supplier. One or more CSs will play the role of the final integrator, assembling the final product (or service) in the chain. The final integrator often has significant influence over the rest of the chain. Logistics and transportation of the flow of supplies is an important and specialist element of the supply chain. The supply chain relies heavily on contracts to ensure that all suppliers meet the requirements of their clients.

**4.3.5.2. Solution**

**Overview**
In this section, we consider how the concepts of the pattern may apply to SoS, and what underlying architectural elements are required for using this pattern in a SoS context. The supply chain is a pattern that sees substantial variation, due to:
- Domain-specific requirements and regulations
- The number of final integrators and final products
- The extent to which suppliers are aware of the final product (some suppliers may simply fulfil standard orders whilst others actively participate in the chain's commitments)
- The standardisation of components and metadata in the domain
- The extent of trust between the CSs, and how much they share their internal dependencies and state

**Structure**
One or more CSs play the role of final integrator, responsible for delivering a manufactured product. Other CSs participate in the SoS by supplying components and/or services, including logistics. The supply chain is not loosely-coupled, with CSs aware of the identity of their neighbours.

Contracts are relied upon to detail the function that will be performed by each CS in the chain, the timing that is required from their and the quality of their output. CSs typically take responsibility for conducting monitoring to ensure that their direct suppliers meet all requirements stipulated. Contractual agreements, negotiations and monitoring may be human activities, or they may be automated. Each CS has individual interfaces typically with its immediate neighbours (suppliers and consumers) in the chain, and with CSs that provide logistics between them. In many SoSs, however, status information is shared more widely throughout the chain, so that integrators downstream can receive information about potential delays as early as possible. This requires availability of interfaces, so that suppliers have a contractual interface with their direct customers, and a status-sharing interface with other members of the supply chain in addition to contractual and/or data sharing interfaces with their immediate logistics suppliers. Security is a concern, as these types of updates may be commercially sensitive, so that only approved partners may receive such data.

Each supplier/integrator CS therefore has the following interfaces:
- Contracts in place with their own suppliers/consumers, stipulating quality, quantity and timing of the goods or services being supplied
- Data-sharing interfaces with their own suppliers/consumers and possibly other CSs downstream
- A flow of goods in and/or out
- A contract with the logistics CS that handles the CS’s own shipments
- Ability to gather data from the logistics CSs that supply or accept shipments from the CS and its immediate neighbours in the chain

**Rationale**
In this subsection, we consider how the solution structure addresses the aims of the pattern.

The aim of efficient product development and delivery is achieved through the use of contracts to tie all CSs into the schedule and quality requirements of the ultimate integrator. The availability of interfaces and data sharing by CSs throughout the chain aids with advance planning, helping to avoid the case that delays propagate down the chain. These interfaces also support greater collaboration between suppliers and integrators.

Easily configurable supply chains are supplied more by open supply chain pattern, and less so by the closed supply chain pattern where contracts, data, manufacturing and/or processes of the CSs may be tailored to the requirements of a particular final integrator and are not easily ported to a new chain. Outsourcing of logistics and the availability of swift movement of goods and sharing of status updates on goods currently in transit means that new suppliers
or integrators from almost anywhere in the world can join the supply chain easily.

**Examples**

As an example we consider an open supply chain centring on the manufacture of clothing. This supply chain incorporates a number of suppliers that provide basic supplies, including fabrics, zips, buttons etc, and a final integrator that is responsible for the final delivery. The components are, for the most part, standardised and can easily be incorporated into other products.

The final integrator accepts a contract to supply items of clothing to be delivered in 6 weeks. The integrator sub-contracts work to a number of third parties located across three continents. Being widely distributed, each supplier and/integrator negotiates its own contracts with its own suppliers in turn, and agrees shipping methods with their selected party logistics suppliers. The supply of some necessary materials – such as the selected fabrics and fabric printing – are stipulated already by the final integrator but for other services the subcontractors have some flexibility.

The various subcontractors share data about current status of orders with the final integrator. In many cases this is automated; the shipments are tagged with RFID tags that are automatically updated as the shipments progress through the manufacturing process.

On completion of manufacture the finished garments are shipped to the retailer using the services of a third party logistics supplier. Data describing current status and location of the shipments is made available by the logistics supplier and tracked by both the subcontractor and the final integrator. If some shipment are delayed the final integrator can alter their own internal scheduling of tasks, to compensate.

4.3.5.3. **Summary**

The supply chain pattern describes the situation when a number of separate companies act as constituent systems, taking responsibility for supplying components towards the ultimate development of a product. This may be relatively simple, with a small number of suppliers and a single integrator, or may be significantly more complex. We characterise a supply chain with standardisation and synchronisation in processes, products and data an open chain, and a chain that relies heavily on bespoke, individually negotiated and adapted products, processes or data interfaces as a more closed chain.
4.3.6. Blackboard Architecture Pattern

4.3.6.1. Introduction

**Background**
Blackboard Systems has been used for many years as described in [Hayes-Roth83] and [Nii86]. The Blackboard architecture pattern was originally developed for software systems and described by [Shaw&Garlan 1996] as an architectural style for data-centred Systems and in [POSA1 1996] as an architectural pattern, here with a more detailed and comprehensive description.

**Aims**
The main aims of this pattern are to support:
- Development of expert or knowledge based systems
- Loose coupling
- Separation of concerns

**Concepts**
The concepts of the blackboard architecture pattern are outlined in Figure 49 below.

![Figure 49 - Blackboard Architecture Pattern Conceptual Model](image)

The main components are the Knowledge Sources, a Blackboard data structure and a Control component.

The Control component can be implemented in the knowledge sources, in the blackboard pattern, in a separate component as shown on the figure or as a combination of these possibilities.

The Blackboard component is a central data store, where elements of the solution space are stored together with control data. The Blackboard component provides an interface for reading and writing data to its internal data structures. Elements of the solution space are written to the Blackboard by the Knowledge Source components. These elements are called Blackboard entries or Hypothesis. Hypotheses can be added as well as removed from the Blackboard.

A Knowledge Source is an independent subsystem or system. Each Knowledge Source is specialized for either solving a part of the overall problem or delivering...
input data to be used by the blackboard algorithm. Each Knowledge Source works independent of the other sources and possible in parallel.

The Control component evaluates the current state of the blackboard by a reading interface and can use this information to coordinate the knowledge sources. The Control component will search for a possible solution to the actual problem, which cannot always be guaranteed.

Figure 50 – Blackboard pattern with possible attributes and operations

As shown on Figure 50 the Knowledge Source has a condition part and an action part (i.e. execCondition() and execAction()). The condition part evaluates the current state of the solution process by inspecting the Blackboard to see if it can make a contribution. The action part produces a result that can be used to update the blackboard state by calling update().

4.3.6.2. Solution

Overview
In this section, we consider how the concepts of the pattern may apply to SoS, and what underlying architectural elements are required for using this pattern in a SoS context.

Structure
This pattern is applicable for directed, acknowledged and possible also for collaborative SoS.

Each constituent system (CS) participating in a SoS can act as a CS-KnowledgeSource generating information which is stored in the SOS-Blackboard component or CS. The SoS-Blackboard component can be implemented by a constituent system and the SoS-Control component as another constituent system. In this way application of this pattern requires one or two CSs acting as the central place for the information (the SoS-Blackboard) and a control system (the SoS-Control) with a specification of the interaction and communication paradigms between these systems. Whether as the knowledge sources can be constituent systems which volunteers to participate in the SoS for obtaining a better solution to the given problem by delivering more input knowledge.
Rationale
In this subsection, we consider how the solution structure addresses the aims of the blackboard pattern stated in section 4.3.6.1.

Development of the expert-based part of the system is mainly located in the SoS-Blackboard component, where the information and hypothesis are stored and modified by the CS-KnowledgeSource components.

The aim of loose coupling is obtained as the CS-KnowledgeSources act independently and do not know the existence of the other sources, but only knows how to access the SoS-Blackboard component or CS. The SoS-Control component is coupled with both the SoS-Blackboard and the CS-KnowledgeSources.

The aim of separation of concerns is obtained by having these three different system roles with each own responsibility. A CS, acting as a knowledge source, can in addition to this have its own more specific purposes, which are not part of the SoS functionality. In addition to this a knowledge source can in principle participate in more than one SoS with different SoS objectives.

Examples
As an example, consider the Healthcare SoS shown in Figure 52.

The SoS consists of three independent constituent systems, one for measuring a person’s blood pressure, another for measuring Electrocardiography (ECG) and a third for measuring Electroencephalography (EEG). These systems are complete systems including distributed measurement units located at the patient site in private homes or in nursing homes and central located data handling and visualization facilities located at healthcare centres e.g. a hospital.
The idea with the Healthcare SoS is to analyse and correlate the different measurements for a given patient and try to set up a diagnose or give warnings in case of a special correlation is found between the inputs from the constituent measurement systems. This analysis and decision functionality is performed in a combination of the SoS-HealthBlackboard and SoS-HealthControl components, which could be located in the same CS. The SoS-HealthControl component can direct the measurement systems to take measurements and also to adjust parameters for the measurements e.g. how often the measurements are wanted or enabling a real-time data stream for making more detailed observations.

A second SoS blackboard example is shown in Figure 53, based on a real-life example called RadarSat-1 [Corkil97], where the Blackboard pattern is used and is described here in the context of a SoS.

RadarSat-1 is an earth-observation satellite launched in 1995. The satellite is equipped with an aperture radar designed to allow end users anywhere in the world to connect to the system and submit measurement requests and to monitor and receive the results from the request when it is executed. The blackboard is used in this system to realize an advanced planning component, which controls the radar measurements. This system has over 140 constraints, which together with the many user requests that have to be taken into account in the planning, makes the planning process very complex.
The SoS-RadarSat-CS is a CS in its own and controls the access and use of the shared resource the build in radar.

Each Client-CS in this SoS is an independent system with its own purpose and local control, but participate in the SoS using a common shared resource, the satellite radar which are shared between the community of RadarSat users. In this way the RadarSat SoS allows each user system to perform its own measurements and experiments with the possibility of sharing the results in the community.

4.3.6.3. Summary

The blackboard pattern can be used to implement SoS, where the SoS problem can be characterized by a certain degree of uncertainty typically found in expert based or fuzzy logic based systems. It can also be applied in SoS where a central knowledge is obtained by several sources i.e. other CS to be used to make a better decision (e.g. a diagnose for a patient) or to be used for an optimal scheduling or planning (e.g. the RadarSat example).
4.3.7. Infrastructure Grid

4.3.7.1. Introduction

Background
In this type of SoS, CSs exchange data in addition to a flow of energy/material. Each CS has the notion of “neighbouring” CSs, with which it may exchange flows of physical items, material and/or data (e.g., energy, flows of traffic, water and sewage). In this type of SoS, the SoS usually is divided into geographical regions, each operated by a designated controller which often has significant autonomy. Each geographical region forms one CS, which is bordered by other regions (CSs). CSs exchange a flow of material (e.g., electricity, traffic) with neighbours, in addition to an exchange of data. Data may be exchanged with any number of CSs contributing towards the SoS, although there is a designated subset of “neighbours” with whom material is exchanged.

The high level goal is typically to ensure large-scale distribution of material (e.g., electric power, water, traffic access) to a large area, and also to offer some level of control over that supply.

The infrastructure grid pattern deals with flows of material or energy, as does the pipe and filter pattern, and the supply chain pattern. However, the infrastructure grid differs from the pipe and filter in the following ways:

- the CSs typically know the identity of their neighbours and are not loosely coupled;
- the flow may be one-way but is often two–way, and in fact may flow in complex networks within and between the CSs;
- CSs may elect to share details of internal state with their neighbours and/or other CSs, and potentially negotiate with neighbours to achieve some collaborative optimal results.

The goal of the infrastructure grid differs from a supply chain. In an infrastructure grid the goods or energy moving through the SoS are typically from large suppliers and distributed to a very large number of smaller consumers; the goal of the supply chain is to assemble a single final product using a number of suppliers and typically just one final integrator.

Aims
The main aims of this pattern are to support:

- Allowing the flow of energy/material around a large area
- Providing planning and management of the flow of material/energy on a very large scale
- Retaining autonomy within distinct geographic areas
- Allowing market competition

Concepts
This type of SoS is composed of some number of constituent systems (CSs). Each CS has the concept of a neighbour, with which it shares a real or metaphorical border, and is typically associated with a geographical region. Each CS is responsible for the distribution of some matter (energy, a physical flow of items, data) within its own borders, and for exchanging this flow with its neighbours. In addition to the flow, metadata about the nature of that flow is exchanged separately. There are separate interfaces required for exchanging data and material.

4.3.7.2. Solution

Overview
CSs exchange a flow of material (e.g., electricity, traffic) with neighbours. There is typically a separate system for exchange of data. There may or may not be a "director" or "leader" associated with the SoS, and any director or leader which does exist may have limited ability to make decisions about overall SoS direction. There is no "hub" via which constituents share material; they exchange materials with "neighbours" directly and therefore this is not a loosely-coupled SoS. Adding or removing CSs is difficult because each CS connects directly to its neighbours. There may or may not be a separate communications hub for sharing data, or the CSs may also exchange data directly with their neighbours in addition to the flow of energy or material.

Examples of this type of SoS include:
- electricity grids
- gas supplies
- water and sanitation system
- road management systems

when viewed at large regional or national levels. The SoS is typically broken into geographical regions with local controllers which have some degree of autonomy; we regard these as the CSs. The CSs exchange flows of energy or material (e.g., power, water, traffic) with their literal geographical neighbours. It's also possible to exchange flows with CSs which are virtual neighbours rather than literal neighbours – e.g., in a water supply system, water flows between neighbouring CSs throughout the network. However, installing a long-distance pipeline to supply water from a CS operating in the far north of country directly to another CS in the far south would allow the two geographically-distant CSs that do not share any geographical border to interoperate as if they are direct neighbours. Similarly, in an electrical grid, a high capacity undersea cable may allow two CSs in different countries to exchange power directly as if they are direct neighbours, even in the absence of any literal geographical border between them.

Structure
Typically one CS will consider a subset of the CSs participating in the SoS to be its neighbours. Data may be exchanged with all of any of the CSs in the SoS, irrespective of whether they are neighbours; data typically facilitates management of the flows that are exchanged. This pattern requires two
interfaces for each CS: an interface that allows the CS to exchange data with other CSs; and a separate interface that allows the CS to exchange the flows with neighbours. For example, in a traffic management system (TMS), a CS that controls flows of traffic in one region will have a control centre that communicates with neighbouring control centres to provide information on current and predicted traffic levels and to agree on traffic management strategies that will be deployed. Separately to this, the flows of traffic move between the CSs independently along roads that cross the CS borders.

The implementation of this pattern may vary, depending on the properties of the material or energy that is exchanged, and the geopolitical structure that surrounds SoS and its geographical area. Variations in the implementation may include the ratio of public/private ownership; the standardisation of both the flow of material and/or the data to manage it; and the regulatory environment. We describe some sub-types of this pattern here:

- Fully decentralised. An infrastructure grid may be fully decentralised if there is no organisation which exerts control or directorship over the SoS emergent behaviour. In this case the CSs exchange data and/or material flows with each other directly and large scale control of the flow is achieved as an emergent behaviour. The fully decentralised pattern is illustrated in Figure 54.

- Partially-decentralised. An infrastructure grid may be partially decentralised if there is an organisation which controls a high proportion of the infrastructure that permits “flow” through different regions. Examples may include electricity grids where the high-voltage power lines within a given country are largely owned by one single party, but where there are a significant number of independent power producers, and/or smaller subsections of the grid controlled by third parties. The partially-centralised pattern is illustrated in Figure 55.

- Data-centralised. An infrastructure grid may be data-centralised if there is a system for collecting and centralising data from the various CSs, even if there is no organisation which acts as an SoS director, or which oversees implementation of SoS goals. This is possible because the CSs in an infrastructure grid have two notions of interface; one for exchanging material; and one for exchanging data. In a data-centralised infrastructure grid, data about the SoS performance and/or about the flows of material in the SoS may be collected or centralised, whilst the CSs still exchange their flows of material directly in a peer-to-peer, decentralised way with no overall director. The data-centralised pattern is illustrated in Figure 56.
It's possible that the infrastructure grid is fully-centralised, with management of the flow of material/energy managed by a single body. An example would be a nationalised utility. Such a system exhibits fewer of the key characteristics that
we associate with an SoS (e.g. autonomy of constituents), although we accept that it may still be able to leverage useful “lessons learned” from such a case study, and that it may have some features and challenges in common with the SoSs we discuss here. However, we don’t discuss further examples of fully centralised infrastructures here.

In many infrastructure grids there is not one single organisation taking responsibility for setting SoS goals and ensuring that all components work towards these goals (this typically only arises in a fully centralised system). Compliance is often assured by two mechanisms:

- Regulatory involvement. Large-scale infrastructures are typically subjected to legally-binding regulation and/or standards. For example, central governments may set desired priorities and goals for traffic management systems across a country, with incentives for compliance (e.g., providing incentives for reducing pollution levels). In another example, water and sanitation supplies must typically comply with legal requirements governing quality of drinking water and pollution levels in rivers and reservoirs. Monitoring of compliance may or may not be assessed by third parties.

- Mutual dependencies. In many cases, the CSs participating in the SoS have different goals. However, sharing a border and exchanging a physical flow of material across that border, they are are also mutually dependent on each other. This ensures that each CS has an incentive to participate in negotiations and data sharing with their neighbours, with an expectation that the commitment is reciprocated.

**Rationale**

The first aim, allowing the flow of energy/material, is enabled by dividing a large area into regions with connected infrastructure; the separate interfaces for infrastructure and data enables this.

The provision of planning and management of the flow of material/energy is supported by the existence of a data-sharing aspect, with separate facilities for gathering sharing data. This allows a constituent to exchange a flow of items, and to exchange data, with different sets of CSs. This means that CSs further down the stream can be alerted to upcoming problems, or requests and reports passed to CSs upstream. If there is a central system for collecting data (i.e., a data centralised infrastructure grid), a global picture of the system can be monitored.

Retaining autonomy within distinct geographic areas is permitted, as typically each area acts as an autonomous CS with control of its own policies (subject to domain-specific regulation). An example is found in traffic management, where different regions with different traffic environments (e.g., urban, rural, trunk roads) can select different traffic management policies and hardware devices to implement traffic management functions within their own borders – e.g., some regions may elect to implement traffic tolls and/or to install speed cameras and other automated enforcement, but others may not. In another example,
independent power producers can elect to generate power in any way they choose (subject to regulations) and still interoperate with a power grid.

Market competition is facilitated by allowing the separate CSs their autonomy, although the extent of market competition will vary according to domain.

**Example**

We provide here examples of the different sub-types of the infrastructure grid pattern.

- **Fully decentralised.** We use as an illustrative example here, the national water/sewage supply in France. Individual municipalities are responsible for implementing supply of water and provision of sewerage services, and they have autonomy in this regard. For example, small individual municipalities sometimes elect to link with others to form larger units that can benefit from larger-scale economic negotiations with suppliers. In some municipalities private contractors are involved in water and/or sewerage, and in others not. Some municipalities choose to contract with one supplier for both services, and others do not. There is no overall hub for sharing data and no overall controller of the SoS (although a large number of separate central agencies have responsibility for cross-cutting aspects of the system, such as regulating water quality or tariffs).

- **Partially-decentralised.** Electricity grids are often partially-decentralised; some direction is desirable over the high-voltage networks of national power transmission systems, for performing planning and load management. We use national grids in the UK and Italy as examples; national grids in these countries are each dominated by a privately-owned company that controls large proportions of the power infrastructure (in the UK this is National Grid plc; in Italy this role is fulfilled by Terna SpA). However, in each country there are additional independent CSs, typically in the form of independent power producers. Terna and the National Grid are capable of making decisions, gathering data and setting goals that heavily affect the overall SoS, although achieving these will often involve collaboration with appropriate independent CSs. Independent suppliers typically compete on quality and/or cost of their service. For this reason there is still a substantial degree of competition and autonomy which may be exhibited by CSs in the SoS, depending on local market.

- **Data-centralised.** As an example, we use the Dutch road network. This national network of roads is divided into separate Traffic Control Centres (TCCs), each responsible for management of roads within its borders. Most TCCs are associated with a limited geographical region, although one is associated with the nationally-distributed trunk network which connects with all the other CSs; the trunk roads are controlled by the national government department responsible for road strategy (Rijkswaterstaat). Because it controls the national trunk road network and is capable of influencing central government policies and regulations, Rijkswaterstaat arguably is able to exert influence over other CSs. Despite this, CSs have some autonomy and may even have differing priorities and goals (e.g., a TCC which is predominantly urban may have
different goals to the trunk network), and each is able to select from a range of possible road management devices and strategies that can be employed within its own borders (subject to Dutch legislation). Each CS exchanges flows of traffic with its neighbours, in a decentralised fashion; the interfaces here are the roads themselves and road users are typically unaware that they have traversed an administrative border as they move through the system. Each CS also has a control centre that monitors traffic and collects traffic data to support decision-making and identification of problems. This data can be shared with other CSs and is ultimately collated centrally by the National Data Warehouse.

4.3.7.3. Summary

The infrastructure grid pattern is commonly associated with the delivery of large-scale civil infrastructure such as electricity and gas supplies, road management, and water and sanitation. It is not a loosely-coupled pattern, with each CS responsible for administering distribution of the given resource over a specific region, and exchanging a flow of resource with its neighbouring regions. This allows regions to assume considerable autonomy, whilst management of the resource is still maintained. We suggest that considerable variation can exist within the implementation of this pattern, depending largely on the domain, regulations affecting it and the degree of centralisation and data sharing.
4.3.8. Reconfigurable Control Architecture (RCA)

4.3.8.1. Introduction

Background
In this type of SoS dynamic architectural reconfiguration of the system is employed as a solution to the SoS problems of:

- continuous evolution
- a challenging runtime environment and
- strongly autonomous and independent nature of the CSs.

Autonomy may cause CSs to cease participating in the SoS, if continued SoS participation no longer aligns with the CS’s individual goals. Alternatively, autonomous third party CSs may unveil unexpected updates that do not take the needs of the SoS into account, resulting in the altering or even removal of functionality upon which the SoS depends. In addition, the distributed nature of an SoS means that connectivity problems are a constant risk; this can lead to a CS becoming completely uncontactable or providing a degraded level of service.

Dynamic reconfiguration is a technique that allows an SoS to adapt and cope with these types of problems. It requires that:

- functions supplied by separate CSs are specified
- alternatives are available
- the SoS has some way of monitoring current availability of those functions and whether they currently meet the minimum specifications needed.

In cases where CSs cease to provide the needed functions or minimum performance that is required at runtime, then the SoS elects to adopt an alternative provider instead, thus dynamically reconfiguring the system.

Aims
The main aims of this pattern are to:

- Allow the SoS to adapt when a CS ceases to provide a function (at some minimum level of performance) by adopting an alternative provider
- Ensure that the situation where architectural reconfiguration is required can be identified
- Ensure that architectural alternatives can be evaluated and the best choice selected

Concepts
This type of SoS is composed of some number of constituent systems (CSs). There exists the notion of an explicit policy that lists certain functions and the minimum performance required for each one. The policy can also detail the conditions under which action may be taken to reconfigure the SoS. Metadata is needed, supplied by each CS to describe the functions they offer. A contractual specification approach is ideal for the RCA; the contract lists the functions that a CS can supply, and details the expected performance that can be expected as long as certain requirements are met. For example, the contract can
state what performance may be expected for a function, as long as the input data meets certain minimum standards. There may or may not be an explicit reconfiguration agent, which is responsible for studying CS performances and determining when architectural reconfiguration is necessary, and if, so, what actions to take.

4.3.8.2. Solution

Overview
Several different solutions are possible to permit dynamic reconfigurable SoS architectures. In all these cases, there needs to be a specification created in advance that identifies the functions upon which the SoS depends, and the minimum performance needed. There will be a default or current configuration for the SoS at any one time; the advance specification should identify possible alternatives for certain system functions. This information is stored in an explicit reconfiguration policy. If the performance of a CS falls below expected requirements, the policy is checked and, if an alternative CS is available offering similar or better performance, then a change in the architecture is enacted to select the new CS as the default provider.

Structure
A number of different approaches to the RCA pattern are possible. We describe possible sub-types of this pattern below:

- Centralised. A centralised RCA consists of a collection of CSs (some currently actively participating in the SoS, some not) and a single configuration Controller which has access to the reconfiguration policy. The Controller takes responsibility for monitoring the CSs’ current performances, and for selecting a new architectural configuration if necessary.

- De-centralised. In this pattern, there is not a single Controller that dictates the current architectural configuration. Instead, the responsibility for monitoring the performance of the current CSs and selecting a new configuration where necessary is shared by distributed Controllers. In this case, the Controllers typically have access to a partial model of the system, and/or are able to monitor only a subset of CSs.
They may have access to reconfigure separate subsets of the SoS with no overlap, or they may each be able to reconfigure the same subset of CSs. In the latter case, an extra policy may be used to determine which Controller’s decisions should be used; this may vary depending on context, or which Controller has the most up-to-date information.

- Uncontrolled. In this version of the pattern, there is no Controller that monitors the behaviour of the CSs. Dynamic reconfiguration is still possible in this situation, if the CSs accept an individual responsibility for monitoring the performance of the CSs with which they directly communicate, and for selecting alternative providers for certain functions if performance falls below acceptable standards. This is implemented by individual CSs only.

The CSs need to have two interfaces for an RCA pattern: interfaces to interact with each other; and an interface to the reconfiguration Controller(s), to allow the Controller(s) to enact a change in the architectural configuration. Dynamic reconfiguration may be more straightforward to enact in a loosely-coupled SoS, or in an SoS where the CSs do not need to know each others’ identities. For example, a Publish-Subscribe or a Blackboard architectural pattern can work well in conjunction with an RCA pattern; when reconfiguration is necessary, the list of CSs currently publishing or subscribing to a given Topic can simply be amended in the Publish-Subscribe pattern, for example. Architectural patterns which are not loosely coupled, or which rely on CSs being aware of each others’ identities, require more complexity to effect a change in the architectural configuration and therefore may be less suited to partnership with an RCA pattern. The exception is in an uncontrolled RCA, where the Controller functionality is implemented internally by each CS.

The contractual specification approach is helpful here for allowing the Controller(s) to evaluate alternatives to a much greater degree, by detailing both the functionality and the expected performance from each CS.

**Rationale**

The first aim, of allowing the SoS to adapt to degraded or absent performance, is achieved, as long as alternative providers are available for key functions that are relied upon. Should the performance of a key CS fall, or the CS withdraws from the SoS altogether, then the SoS can adapt to cope by selecting a new architectural configuration which relies upon other providers (effectively replacing existing CSs with new CSs).

The Controller(s) take(s) responsibility for monitoring performance and identifying when reconfiguration is necessary. This does rely on a pre-written specification which describes the necessary performance for each CS, and the availability of metadata which describes services provided by each CS. The availability of metadata and contractual specifications that describe the CS functions permits the Controller(s) to evaluate the selection of CSs and select and alternative configuration when necessary.
Example
We provide here examples of the different sub-types of the RCA pattern.

- **Centralised.** In a centralised RCA pattern, there is a single controller that takes responsibility for monitoring the SoS and effecting reconfiguration. As an example, consider a traffic management system that monitors traffic on a trunk network, and which can dynamically add or remove CS functions. The traffic control centre (TCC) aims to achieve some optimal throughput on the network, and a number of different functions are available to achieve this, implemented by different CSs. The TCC might, for example, to begin by regulating the rate at which vehicles are permitted to join the network, using a system of ramp meter devices to slow down the number of vehicles entering the system, and so it adds the ramp metering functionality to the SoS. However, if the ramp meters do not work as expected (e.g., some ramp meters are unavailable or lose power), the TCC can elect to add in other functions, implemented by other CSs, to try and achieve the same result. For example, it can select to add dynamic speed limits, dynamic message boards, or dynamic lane closures/openings, to alter the throughput and achieve the desired goal. The TCC stores policies that list these strategies and the conditions under which they should be enacted (e.g., perhaps changing the speed limit should only be considered when all other strategies have failed to achieve the desired result, and the gap between current and desired throughput is greater than n).

- **De-centralised.** A de-centralised RCA pattern sees two or more Controllers taking responsibility for evaluating and reconfiguring the SoS. For example, we consider a system that is responsible for managing traffic in a city centre. The SoS has a number of goals, including reducing air pollution within the urban area and achieving optimum traffic movement. There are several CSs gathering input data, including distributed pollution sensors at locations around the city, and information on traffic movements, gathered from traffic sensors around the city. There are two controllers capable of reconfiguring the system: one makes decisions based on pollution data; and the second makes decisions based on traffic data. Both controllers are capable of adding new functions/CSs to the SoS, and/or dispatching human officers to locations around the city, in order to enact traffic management strategies that will help to achieve their goals. In many cases both controllers will wish to enact some strategy, one aiming to reduce air particles and one to speed up traffic movements. A policy exists which determines which Controller has priority in situations where both wish to reconfigure the system. In our example, we assume that the Controller responsible for reducing pollution has priority in cases where air pollution exceeds some extreme levels; in cases where pollution is slightly elevated but traffic is very slow, then the traffic management Controller has priority, and so on.

- **Uncontrolled.** An implementation of the uncontrolled RCA pattern would see each CS taking separate responsibility for checking performance of its peer providers, and hunting for better alternatives when necessary. An example of this can be seen in a computer network that relies on a number of autonomous routers for handling message routing. Each
router maintains a table of its peers with whom it can communicate, and their current response time. If one of its peers becomes unavailable, the router selects an alternative peer for passing on messages, after studying the list of current performance times and its (limited) information on which routers are suitable candidates for certain routes. This results in emergent behaviour that the network dynamically adopts new routes if some of the nodes fail.

4.3.8.3. Summary

The Reconfiguration Control Architectural pattern is a pattern that allows an SoS to implement some degree of dynamic architectural reconfiguration. This can vary, depending on the degree of control exerted and whether this is centralised or decentralised. The pattern relies on the presence of separate interfaces for CS functionality and/or allowing a configuration Controller(s) to enact a given strategy. An explicit policy is required to govern the reconfiguration; this should be provided in advance, and detail the functions that the SoS requires, the quality needed and the thresholds at which alternative should be considered. A contractual specification approach is helpful here, for allowing the Controller(s) to evaluate alternatives effectively.
4.4. Enabling Patterns

Enabling patterns are specific constructs of modelling elements whose combination and subsequent use enables a number of systems engineering applications. Such patterns do not define architectures for systems but allow aspects of a system’s architecture to be modelled in a consistent manner. A pattern for the definition of interfaces or one used to ensure traceability throughout a model of a system are examples of enabling patterns.

The following are the existing enabling patterns identified for systems engineering:

- Context
- Interface definition
- Life cycle
- Measurement
- Ontology
- Process
- Specification
- Test case
- Traceability

Some key aspects of SoS development are interfaces within and between systems (both SoSs and CSs), traceability and its importance to impact and change analysis and testing to ensure verification and validation of both the CSs and the SoS that they make up. The life cycles that systems (SoSs and CSs) go through and their evolution through epochs of time are also an essential aspect of systems engineering. Certain interfaces may be based on defined contracts between the parties involved. Therefore, the following enabling patterns are described:

- Interface Definition Pattern
- Test Pattern
- Traceability Pattern
- Life Cycles Pattern
- Epoch (System evolution) Pattern
- Contract Pattern

4.4.1. Interface Pattern

**Background**

Interfaces form an integral part of any systems model and define a contract between system elements, whether those elements are physical or are realised in software. They capture the nature of the interactions between those elements, specifying both what can be transferred between the elements and how such transfers take place. Defining interfaces correctly is essential if the system elements are to work properly with each other.

*Note on the use of the term 'System Element' in this pattern:* This pattern uses the term ‘System Element’ throughout, defining a pattern for modelling
Interfaces between system elements. This pattern can be used for modelling interfaces at both the system and system-of-systems level. When used at the system level, then a system element will be a part of the system (for example, a sub-system or component or assembly, depending on the system hierarchy being employed). When used at the system-of-systems level, then a system element will be a constituent system of the SoS under consideration. Put simply an SoS is a system whose system elements are constituent systems.

4.4.1.1. Pattern Aims

This pattern is intended to be used as an aid to the definition of interfaces. The main aims of this pattern are shown in the Architectural Framework Context View (AFCV) in Figure 58.

![AFCV Pattern Aims - Interface Definition](image)

Figure 58 - Architectural Framework Context View showing Interface Definition aims

The key aim of the Interface Definition Pattern is to 'Identify Interfaces', the identification of interfaces and their relation to the system elements that use them and the ports that expose them. This use case includes further use cases:

- 'Define operations' – defining the interfaces in terms of the operations they may provide.
- 'Define flows' – defining the interfaces in terms of the flows of data, material, energy, personnel etc. that take place across an interface.
- 'Identify connections' - identification of the connections between ports that expose interfaces and of the interface connections that take place across those port connections.
• 'Define protocols' - definition of any protocols to which an interface or port must conform.
• 'Identify scenarios' - identification of typical scenarios showing how interfaces are used.

This pattern can be used for interfaces between physical system elements as well as systems elements realised in software.

4.4.1.2. Concepts

The main concepts covered by the Interface Definition Pattern are shown in the Ontology Definition View (ODV) in Figure 59.

Key to this pattern is the concept of the 'Interface'. An 'Interface' has a 'Direction', which may take the values "in", "out" and "inout". The 'Direction' property of an 'Interface' shows the direction in which the 'Interface' operates from the point of view of the 'Port', owned by a 'System Element', which exposes the 'Interface'. Two types of 'Interface' exist, the 'Service-Based Interface' and the 'Flow-Based Interface'. Service-Based Interfaces are used to represent those Interfaces that are operation or service-based such as are typically found in software-intensive Systems. Flow-Based Interfaces are used to represent those Interfaces that transfer data, material, energy, personnel etc. between System Elements. For example, an Interface between a fuel pump and an engine would be represented by a Flow-Based Interface.

Each 'Interface' is described by an 'Interface Definition'. This defines the operations of a Service-Based Interface and the items transferred by a Flow-Based Interface. These operations and flows use 'Flow Types'. For example, an Interface for a transmitter may have a “transmit” operation that takes a power level parameter. The type of this parameter would be defined using a Flow Type. Similarly, the type of fluid pumped by a pump would be described by a Flow Type.
A 'Port' represents the interaction points between one or more 'System Element' and may represent the concept of a software Port or a physical Port, such as the connector for the fuel line on a car engine fuel pump. Ports are connected to each other via a 'Port Connection'. A fuel rail taking fuel from the fuel pump to fuel injectors in a car engine would be represented by a Port Connection.

Interfaces can be connected together, but only if both ends of the connection are described by the same Interface Definition and have complementary Directions (or if at least one of the ends has Direction "inout"). Such a connection is modelled as an 'Interface Connection' that takes place across a 'Port Connection'. For example, the transfer of fuel from a pump to an engine through a fuel rail would be modelled as an Interface Connection.

Finally, Ports and Interfaces may conform to one or more 'Protocol' that describe and control how the Port and Interface behaves.

### 4.4.1.3. Pattern Viewpoints

The Interface Definition pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 60.

The Interface Definition pattern defines five Viewpoints for the definition of Interfaces:

- The 'Interface Identification Viewpoint' is used for the identification of Interfaces and Ports and their relation to the System Elements that use them.
- The 'Interface Connectivity Viewpoint' used to show how Interfaces and Ports are connected.
- The 'Interface Definition Viewpoint' is used for the definition of Interfaces in terms of the operations they may provide and the flows of data, material, energy, personnel etc. that take place across an Interface.
• The 'Interface Behaviour Viewpoint' is used for the identification of typical scenarios showing how Interfaces are used.

• The 'Protocol Definition Viewpoint' is used for the definition of any Protocols to which an Interface or Port must conform.

Each of these Viewpoints is described in more detail in the following sections. For each Viewpoint an example is also given.

4.4.1.4. Rules

Six rules apply to the five Interface Definition Viewpoints, as shown in the Rules Definition View (RDV) in Figure 61:

Note that the six rules shown in Figure 61 are the minimum that are needed. Others could be added if required.

4.4.1.5. Interface Identification Viewpoint (IIV)

The aims of the Interface Identification Viewpoint are shown in the Viewpoint Context View in Figure 62.
The main aim of the Interface Identification Viewpoint is to ‘Identify Interfaces’, which includes:

- ‘Identify system elements using interfaces’ – identification of the System Elements and the Ports that they own.
- ‘Identify exposing ports’ – identification of the Ports that expose the Interfaces and the Interfaces that they expose.

**Description**

The Viewpoint Definition View (VDV) in Figure 63 shows the Ontology Elements that appear on an Interface Identification Viewpoint.

Also shows Interface Definitions by name, but does not show their content.
The Interface Identification Viewpoint shows System Elements and the Ports that they own. The Interfaces exposed by Ports are also shown, along with the names of the Interface Definitions that describe them.

**Example**

Example Views that conform to the Interface Identification Viewpoint are shown in Figure 64 and Figure 65.

The Interface Identification View in Figure 64, realised as a SysML `internal block diagram`, shows two System Elements, namely a 'Pump Controller' and a 'Pump'. Each of these has a Port shown by the small squares on the right-hand edges of the 'Pump Controller' and the 'Pump'. The two Ports have their names ('Controller Output' and 'Controller Input' respectively) and type ('USB') shown.

The Ports both expose an Interface that is described by the 'PumpIF' Interface Definition. For the 'Pump Controller', the Direction of the Interface is “out”, as shown by the use of the SysML `required interface` notation (the “cup”). For the 'Pump', the Direction of the Interface is “in”, as shown by the use of the SysML `provided interface` notation (the “ball”).

From a SysML point of view these are both “inout” Interfaces, as a `required interface` in SysML can accept `return values` and a `provided interface` can send such values. However, from the point of view of the initiation of the communication across the Interface, the direction is “out” on the 'Pump Controller' and “in” on the 'Pump'.

This view, along with the Interface Connectivity View in Figure 68 and the Interface Definition View in Figure 72, fulfils Rule ID1.
The Interface Identification View in Figure 65, realised as a SysML *internal block diagram*, shows three System Elements: 'Pump', 'Tank' and 'Hole'. Each of these exposes a number of Flow-Based Interfaces via the various Ports that each own. These Ports are shown using SysML *flow ports*, the small squares containing arrow heads, which indicate the directionality of the Interfaces.

The 'Pump' has an *out* Flow-Based Interface named 'ToSupply' that can transfer 'Liquid', an *in* Flow-Based Interface named 'FromSupply' that can receive 'Liquid' and an "inout" Flow-Based Interface named 'Outflow/Inflow' that is of type 'LiquidFS'.

The 'Tank' has an “in” Flow-Based Interface named 'InFlowValve' that can receive 'Liquid' and an “out” Flow-Based Interface named 'OutFlowValve' that can transfer 'Liquid'.

Finally, the Interface to the 'Hole' is modelled as an un-named “inout” Flow-Based Interface that is of type 'LiquidFS'.

This View, along with the Interface Connectivity View in Figure 69 and the Interface Definition View in Figure 72, fulfils Rule ID1.
4.4.1.6. Interface Connectivity Viewpoint (ICV)

The aims of the Interface Connectivity Viewpoint are shown in the Viewpoint Context View in Figure 66.

The main aim of the Interface Connectivity Viewpoint is to 'Identify Interfaces' with an emphasis on 'Identify Connections', showing the connections between Ports and the Interface Connections that take place across the Port Connections.

**Description**

The Viewpoint Definition View (VDV) in Figure 67 shows the Ontology Elements that appear on an Interface Connectivity Viewpoint.
The Interface Connectivity Viewpoint shows System Elements and the Ports that they own. The Port Connections between Ports are shown together with the Interfaces exposed by Ports and the Interface Connections between these Interfaces.

The names of the Interface Definitions that describe each Interface are shown, but the content of the Interface Definitions is not shown.

The Interface Connectivity Viewpoint can be thought of as containing the Interface Identification Viewpoint. However, not all the Ports and Interfaces shown on an Interface Identification View (an instance of an Interface Identification Viewpoint) need be connected, or, alternatively, they can be connected in different ways depending on different configurations of System Elements. In such cases then both diagrams are needed, otherwise the Interface Identification Viewpoint can be omitted.

**Example**

Example Views that conform to the Interface Connectivity Viewpoint are shown in Figure 68 and Figure 69.
The Interface Connectivity View in Figure 68, realised as a SysML *internal block diagram*, shows two System Elements, namely a 'Pump Controller' and a 'Pump'. Each of these has a Port shown by the small squares on the right and left edges of the 'Pump Controller' and the 'Pump' respectively. The Port Connection between these two Ports is shown, and use has been made of SysML *stereotypes* to annotate the connector to show that physically this is a USB cable as shown by the «USB Cable» stereotype.

Both Ports expose an Interface that is described by the 'PumpIF' Interface Definition. The Interface Connection between them is shown by the connection of the SysML “cup” and “ball” notation used to indicate the Interfaces.

The types of the two Ports have been omitted but these could have been shown (as 'USB Port') if required.

This View, along with Figure 64 and Figure 72, fulfils Rule ID1.
The Interface Connectivity View in Figure 69, realised as a SysML internal block diagram, identifies three Flow-Based Interfaces between three System Elements. There are two between the 'Tank' and the 'Pump' and one between the 'Pump' and the 'Hole'.

The Port Connections between the Ports are shown, and use has been made of SysML stereotypes to annotate the these connectors to show the type of physical connection. There are two '3" 11 Gauge Non-hardened Lay-down Pipe' between the 'Tank' and the 'Pump' and one '4" 11 Gauge Non-hardened Lay-down Pipe' between the 'Pump' and the 'Hole'.

The SysML flow ports used on the diagram show the directionality of the Interfaces. The Interfaces between the 'Pump' and the 'Hole' have a Direction of "inout". The 'Pump' can pump into and out of the 'Hole' through a single pipe. The two Interfaces between the 'Tank' and the 'Pump' are each uni-directional. From the point of view of the 'Tank' it has an Interface with a Direction of "out" to supply 'Concrete' to the 'Pump' via its 'OutFlowValve', and an Interface with a Direction of "in" to receive 'Concrete' from the 'Pump' via its 'InFlowValve'. The SysML item flows carrying 'Concrete' across the connectors define the Interface Connections.

The types of the Ports have been omitted but these could have been shown, as in Figure 65, if required.

This View, along with Figure 65 and Figure 72, fulfils Rule ID1.
4.4.1.7. Interface Definition Viewpoint (IDV)

The aims of the Interface Definition Viewpoint are shown in the Viewpoint Context View in Figure 70.

The aim of the Interface Definition Viewpoint is to 'Identify interfaces', with an emphasis on:

- 'Define operations' – definition of Interfaces in terms of the operations they provide.
- 'Define flows' - definition of Interfaces in terms of the flows of data, material, energy, personnel etc. that take place across an Interface.

**Description**

The Viewpoint Definition View (VDV) in Figure 71 shows the Ontology Elements that appear on an Interface Definition Viewpoint.
The Interface Definition Viewpoint contains a number of Interface Definitions, together with the Flow Types of the items passed across the Interfaces that are described by the Interface Definitions.

Note that this Viewpoint does not show Interfaces, but concentrates on the descriptions of Interfaces through the Interface Definitions and Flow Types that describe them.

*Example*
An example View that conforms to the Interface Definition Viewpoint is shown in Figure 72.
This Interface Definition View, here realised as a SysML block definition diagram, shows three Interface Definitions.

The first Interface Definition is 'PumpIF' modelled using a SysML interface block. 'PumpIF' is an example of an Interface Definition for a Service-Based Interface. It defines a number of services that the Interface provides, realised here using SysML operations, an example of which is the 'start' service. It also defines a single property, 'CurrentDirection', which is used to store information about the state of the Interface when it is in use.

'DirectionType', 'Boolean' and 'PowerLevel' are all examples of Flow Types that are used by the 'PumpIF' as the types of parameters of the three services 'start', 'stop' and 'reverse' and of the 'CurrentDirection' property. The “uses” relationship is made explicit through the use of the stereotyped SysML dependency.

The second Interface Definition is 'LiquidFS' modelled using a SysML flow specification. 'LiquidFS' is an example of a Flow-Based Interface. It defines an
Interface in terms of items that can flow across the Interface. In this case, it shows that 'Liquid' can flow in and out of the Interface.

'Liquid' and its sub-types of 'Concrete', 'Oil' and 'Water' are, again, examples of Flow Types that are used by the 'LiquidFS' Interface Definition.

The third Interface Definition is given by 'Liquid' (and its sub-types). 'Liquid', as well as being an example of a Flow Type, is an Interface Definition in its own right, describing four of the Interfaces that appear on the Interface Identification View in Figure 65.

This View, along with the Interface Identification Views in Figure 64 and Figure 65 and the Interface Connectivity View in Figure 68 and Figure 69, fulfils Rule ID1. It also fulfils Rule ID3 and Rule ID4.
4.4.1.8. Interface Behaviour Viewpoint (IBV)

The aims of the Interface Behaviour Viewpoint are shown in the Viewpoint Context View in Figure 73.

The main aim of the Interface Behaviour Viewpoint is to 'Identify interfaces' with an emphasis on 'Identify scenarios', identification of typical scenarios showing how interfaces are used.

**Description**

The Viewpoint Definition View (VDV) in Figure 74 shows the Ontology Elements that appear on an Interface Behaviour Viewpoint.
The Interface Behaviour Viewpoint shows a number of System Elements interacting with each other via the services and items transferred across the Interfaces between the System Elements. Since these interactions are governed by the Interface Definitions and associated Flow Types that describe each Interface, the Interface Behaviour Viewpoint indirectly shows elements of the Interface Definitions and Flow Types.

**Example**

Example Views that conform to the Interface Behaviour Viewpoint are shown in Figure 75, Figure 76 and Figure 77.

The Interface Behaviour View in Figure 75, here realised as a SysML sequence diagram, shows the interactions between two System Elements, the 'Pump Controller' and the 'Pump'.

As shown in Figure 68, all interactions between the 'Pump Controller' and the 'Pump' must conform to 'PumpIF'. That is, they must conform to the Interface Definition that is described on the Interface Definition View in Figure 72. The diagram shows messages corresponding to the 'start', 'stop' and 'reverse' services shown on the Interface Definition View being sent from the 'Pump Controller' to the 'PumpIF'.

This View, along with the Interface Definition View in Figure 72 and the Interface Connectivity View in Figure 68, fulfils Rule ID5.

This example illustrates a simple Interface without any governing Protocol. An example of an Interface Behaviour View for the 'Pump Controller' and 'Pump'
where the 'PumpIF' does conform to a governing Protocol can be seen in Figure 76.

![IBV Pump Controller to Pump - Normal Single Cycle Operation Showing PumpIF](image)

The Interface Behaviour View in Figure 76, here realised as a SysML sequence diagram, shows two System Elements, the 'Pump Controller' and the 'Pump', connected together by the 'PumpIF', an example of a Service-Based Interface.

As shown in Figure 68, all interactions between the 'Pump Controller' and the 'Pump' must conform to 'PumpIF'. That is, they must conform to the Interface Definition that is described on the Interface Definition View in Figure 72. That this is so can be seen in the diagram where SysML messages corresponding to the 'start', 'stop' and 'reverse' services shown on the Interface Definition View can be seen being sent from the 'Pump Controller' to the 'PumpIF'.

In this example the 'PumpIF' has an internal Protocol (see Figure 80 below) that translates the 'start', 'stop' and 'reverse' messages that it receives into a series of 'prime', 'pump', 'stopPump', 'pumpReverse' and 'flush' signals that it forwards to the 'Pump'.

Although the 'PumpIF' is shown as a separate SysML lifeline in the diagram, this has been done to emphasise the internal behaviour of the Interface and to make explicit the translations that are performed by its governing Protocol. The 'PumpIF' should not be thought of as being separate from the 'Pump'; it simply defines a set of services provided by the 'Pump'. When implementing this System, this aspect of the behaviour of 'PumpIF' could be implemented in software running on the 'Pump'. The 'PumpIF' is acting as a “wrapper” to the
'Pump', providing a simple set of three services that can remain constant if the internal operation of the 'Pump' changes or that can be used on pumps with different behaviour. For example, if a self-priming pump that could go directly from pumping to pumping in reverse without stopping was to replace the existing 'Pump', then provided it exposed the same 'PumpIF' no changes would be needed to the type of 'Pump Controller' used. The changes would be reflected in the Protocol for the 'PumpIF' as implemented by the new 'Pump'.

If an Interface is of a simpler kind, without any governing Protocol, then often it will not be explicitly shown on an Interface Behaviour View. For example, the 'Pump Controller' could be shown communicating directly with the 'Pump', sending 'start', 'stop' and 'reverse' messages directly to it, as long as the 'Pump' could handle such messages without the need for a Protocol to convert them into the 'prime', 'pump' etc. signals.

This View, along with the Interface Definition View in Figure 72 and the Interface Connectivity View in Figure 68, fulfils Rule ID5.

The Interface Behaviour View in Figure 77, modelled here using a SysML sequence diagram, shows an example of System Elements interacting via Flow-Based Interfaces.

The diagram shows three System Elements: 'Tank', 'Pump' and 'Hole'. As shown in Figure 69, they interact according to the 'LiquidFS' and 'Liquid' Interface Definitions that are defined on the Interface Definition View in Figure 72. The
scenario shown in the View corresponds to that shown in the corresponding Service-Based Interface IBV seen in Figure 76, but from the point of view of the items flowing between the various System Elements rather than the services invoked by one on another.

It should also be noted that, in this scenario, it is 'Concrete' that is flowing between the various System Elements. Because of the way the Interfaces are defined any of the defined sub-types of 'Liquid', such as 'Concrete', 'Oil' or 'Water' could have been used.

Also, although the behaviour of the two types of Interface has been shown on separate Interface Behaviour Views, there is nothing to prevent these two diagrams being combined in to a single IBV showing the behaviour of both types on a single diagram.

This View, along with the Interface Definition View in Figure 72 and the Interface Connectivity View in Figure 68, fulfils Rule ID5.

4.4.1.9. Protocol Definition Viewpoint (PDV)

The aims of the Protocol Definition Viewpoint are shown in the Viewpoint Context View in Figure 78.

The main aim of the Protocol Definition Viewpoint is to 'Identify interfaces' with an emphasis on 'Define protocols', defining any protocols to which an interface or port must conform.

*Description*

The Viewpoint Definition View (VDV) in Figure 79 shows the Ontology Elements that appear on a Protocol Definition Viewpoint.
The Protocol Definition Viewpoint contains the Protocol for an Interface or Port. Such Protocols define the behaviour governing the Interface or Port. For example, an interface to a 'Pump' ('PumpIF', say) might ignore 'reverse' messages when the 'Pump' is pumping until the 'Pump' is first stopped by sending a 'stop' message to the 'PumpIF'.

Each Interface or Port can have multiple Protocols governing their behaviour. For example, an intelligent 'PumpIF' could follow a different control Protocol depending on the type of 'Pump Controller' connected to it.

Protocols often make use of concepts such as events and signals. Such concepts have been deliberately omitted from the Protocol Definition Viewpoint (and the ODV in Figure 59) as they are dependent on the representation adopted for the realisation of the Viewpoint.

**Example**
An example View that conforms to the Protocol Definition Viewpoint is shown in Figure 80.
The Protocol Definition View, here realised as a SysML state machine diagram, describes the Protocol to which the 'PumpIF' must conform.

The 'PumpIF' provides three services: 'start', 'stop' and 'reverse'. It must convert invocations of these services into the relevant signals to be issued to the 'Pump' to which it provides an Interface.

In this example, an invocation of the 'start' service must be converted into a 'prime' and then a 'pump' signal to the 'Pump'. Similarly, 'reverse' is converted into 'stopPump' and 'pumpReverse' signals and 'stop' into either a 'flush' and then a 'stopPump' signal or just a 'stopPump' signal depending on whether an emergency stop is being requested.

In order to be able to correctly handle a 'reverse' service request, the 'PumpIF' must maintain information on which direction the 'Pump' is running. This is held in the 'CurrentDirection' property which is given the value 'Forward' or 'Reverse' as appropriate.

An observation that can be made about this diagram is that it takes no account of the 'powerLevel' parameter passed in with the 'start' and 'reverse' service calls (and shown on the Interface Behaviour View in Figure 76). Perhaps this implementation of the 'PumpIF' Protocol is intended to be used with a 'Pump' that does not take a 'powerLevel'. If it does take a 'powerLevel', then the handling of 'powerLevel' should be added to this View and this would also require changes to the Interface Behaviour View in Figure 76.

This View fulfils Rule ID2 and Rule ID6.
4.4.1.10. Extensions to the Interface Pattern

There are a number of possible extensions that can be made to the Interface Definition pattern. Four such extensions are briefly considered here:

1. Timing
2. Security
3. Quality of Service
4. Design by Contract

Each of these extensions is a large subject and could be a pattern in their own right. For this reason, the coverage here will intentionally be brief. The aim is not to fully explore these areas, but rather to give a flavour of how aspects of these extensions could be considered in the existing Interface Definition pattern.

**Timing**

Often a key aspect in the definition of the Interfaces between System Elements is that of timing. Timing is often a contentious issue in systems engineering, particularly whether or not the System is a real-time system (and indeed, often what is meant by real-time). Extensions exist to the Unified Modeling Language, on which the Systems Modeling Language is built, designed specifically to allow the modelling of real-time Systems. One such extension is MARTE (Modeling and Analysis of Real-Time and Embedded Systems), managed by the Object Management Group [MARTE 2011].

Two aspects of timing that form a simple extension to the Interface Pattern are:

1. Timing constraints on interactions between System Elements.
2. Timing constraints within a System Element governing response time to a receipt of an item or service request from an external System Element.

Both these types of timing constraints can represented on the Interface Behaviour View. Examples of both can be seen in Figure 81, an extended version of Figure 76.

The first type of timing constraint can be seen on the interactions between System Elements, such as the 'start (90%)' message between 'Pump Controller' and 'PumpIF'. The message has been annotated with the *timing constraint* '{<= 10ms}' to indicate that this interaction must take no longer than 10ms. What is not clear from this *timing constraint* is whether this is a maximum time (i.e. 10ms is never to be exceeded) or an average time (i.e. 10ms must be the maximum transmission time, on average). The latter case then raises the question of how such an average is calculated. Other meanings could also be given to such *timing constraints*, such as expected time (i.e. the system engineers expect this interaction to take at most 10ms) or observed time (i.e. the average observed time in test). Such information can be added to the diagram if required. This has been done for the 'reverse (90%)' message between 'Pump Controller' and 'Pump'. The *timing constraint* now shows that the maximum time that can
elapse between the sending and receipt of the message is 12ms, that the average desired transmission time is 10ms and that the expected time is 5ms.

The second type of timing constraint is shown using the left-most vertical double-headed arrows in Figure 81. For example, the arrow labelled with the constraint '{<= 500ms}' at the top of the column shows that after receiving the 'start( 90% )' message the 'PumpIF' has at most 500ms in which to send the 'prime' message to the 'Pump'.

While not specifically associated with the Interfaces, additional timing information has been added to show the timing associated with behaviour carried out within a System Element. These are the right-most vertical timing constraints. For example, Figure 81 shows that the 'prime' operation, once started, will take between 10 and 30 seconds.

The same observations about the meaning of such timing constraints apply as discussed for those on the interactions between System Elements; the diagram could be similarly annotated to distinguish between maximum, average, expected and observed times.

As well as specifying constraints on particular interactions, an annotated Interface Behaviour View like that in Figure 81 is also useful for understanding the end-to-end timing constraints that must hold for the system. From Figure 81 it is possible to see that a maximum time of 520ms can elapse from the sending of the 'start( 90% )' message by the 'Pump Controller' until the 'Pump' starts to 'prime' itself. As well as summing the timings across the diagram, the timings can...
also be summed down the lifelines (Figure 81 would need two additional timings added to allow this to be done: the duration of the 'pump' and 'pumpReverse' operations). Such calculations are easily automated in SysML tools and provide valuable additional information when using Interface Behaviour Views to help define test cases for verification and validation testing.

**Security**
The topic of security is a vast one, and includes such topics as:

- Identity management [Arabo et al 2011]
- Security policies [Bodeau 1994]
- Data flow risks, the dynamic configuration of systems and unknown runtime configurations [Kennedy et al 2010]
- The heterogeneous nature of systems [Shone et al 2011]

Any of these topics could be considered as suitable security extensions to the Interface Pattern. However, given the complexity and issues surrounding security, it would be an excellent candidate to be a pattern in its own right.

The intention of this section is not to investigate these topics but to show how the Interface Pattern can, through simple extensions, be extended to cover some common aspects of security as it relates to interfaces.

Three aspects of security that form a simple extension to the Interface Pattern are:

2. Security limits on the Flow Types that can flow across Interface Connections.

Each of these three aspects is discussed below.

Figure 82 presents an example of how security limits on Port and Interface Connections can be represented on an Interface Connectivity View. Both the Port Connection and the Interface Connection have the «secure» stereotype applied and each have values assigned to the associated 'level' tag.
The diagram is intended to show that maximum security level of the Port Connection is 15 (on a scale of 1 to 20, arbitrarily defined for this example) whereas the Interface Connection is shown to have a maximum level of 10. Thus the Interface Connection can be allowed to take place across the Port Connection. If the levels had been switched, then the Interface Connection would be exceeding that of the Port Connection meaning that any transactions taking place across that Interface could be unsecure. The relationship between these two levels gives rise to a new Rule (ID7). Additional Rules are given at the end of this section.

The second security aspect is illustrated in Figure 83, an enhanced Interface Definition View. Firstly, the 'MsgType' block (representing a Flow Type) has the «secure» stereotype applied to indicate that 'MsgType' can be assigned a security level when in use. Secondly, the 'msg' parameter (of type 'MsgType') of the 'handleMsg' operation of the 'MsgIF' interface block (representing an Interface Definition) also has the «secure» stereotype applied, as well as having a value set for the 'level'. This defines the maximum possible security level that the parameter can take when the Interface defined by the Interface Definition is used. Additional Rule ID8 arises from this.
The final security aspect, namely that of the security limits on Interfaces when in use are illustrated in Figure 84 and Figure 85.

Figure 84 shows an instance, 'm1', of a block, typed by 'MsgType' (defined above in Figure 83). However, where Figure 83 indicates that 'MsgType' is marked as 'secure', the instance 'm1' has the 'level' of security shown.

It should be noted here that, although the diagram in Figure 84 is a SysML structural diagram (it is a block definition diagram showing instances of blocks) it has been marked here as being an Interface Behaviour View which earlier was shown to be realised using a SysML sequence diagram, which is a behavioural diagram. This was done here primarily to save space; in a full definition of the security extension to the Interface Definition Pattern it is likely that an additional Viewpoint would be defined that would be used to model actual parameters etc. Since this hasn't been done here and since actual parameters are associated with examples of an Interface in use, the diagram in Figure 84 has been treated as an extension to the Interface Behaviour View.
Defining the security of Flow Types applicable to them when in use allows both the defined and the actual security levels of items flowing across Interfaces to be modelled, as shown in Figure 85.

The two comment notes attached to the message between 'Tx' and 'Rx' show the defined and actual security levels for the parameters of the 'handleMsg' message exchanged. An additional Rule, ID9, arises as a result of this.

As discussed, the security enhancements give rise to additional Rules that must hold. These could be added to the RDV shown in Figure 61 or shown on an additional RDV. Here a textual representation is used to emphasises that although SysML has been used throughout the Interface Definition Pattern for the definition of the pattern and for most of the examples, a realisation in a notation other than SysML is valid.

- Rule ID7: If a Port Connection is marked as secure, then any Interface Connections that take place across the Port Connection must also be marked as secure and the security level of the Interface Connection must be ≤ that of the Port Connection.
- Rule ID8: If an Interface Connection is marked as secure, then any Flow Type that can pass across that Interface Connection (e.g. a parameter of an operation for a Service-Based Interface or an item defined by a Flow Type that flows across a Flow-Based Interface) must also be marked as secure and the security level of the Flow Type must be ≤ that of the Interface Connection.
- Rule ID9: If an Interface Connection is marked as secure, then any Flow Type that does pass across that Interface Connection (e.g. an item passed as a parameter of an operation for a Service-Based Interface or an item defined by a Flow Type that flows across a Flow-Based Interface) must also be marked as secure and the security level of the passed Flow Type must be ≤ that of the defined Flow Type.
Essentially, the Rules say the following:

- **Rule ID7:** An Interface Connection can not be more secure than the Port Connection it is taking place across.
- **Rule ID8:** Nothing that is defined as being able to be transferred across an Interface Connection can be more secure than the Interface Connection.
- **Rule ID9:** Nothing that does pass across an Interface Connection can be more secure than the corresponding definition of what can pass across.

Figure 82 conforms to Rule ID7: the Interface Connection is marked as secure and with a lower level than the Port Connection that it takes place across.

Figure 83 conforms to Rule ID8: the parameter being passed across the Interface Connection through the Interface defined by the 'MsgIF' Interface Definition is marked as secure (as is its corresponding Flow Type, 'MsgType'). The level for this parameter is ≤ that of the Interface Connection.

Figure 84 and Figure 85 conform to Rule ID9: the actual parameter passed across the Interface Connection through the 'MsgIF' is secure and its level is ≤ its corresponding parameter definition.

That is, the Port Connection can handle items with a maximum security level of 15 but the Interface Connection taking place across the Port Connection has been marked as operating at the lower security level of 10, which is acceptable. The maximum security level of anything that will be passed across the Interface Connection is also 10, so no attempt will be made to exceed the security limit of the Interface Connection. Finally the maximum level of anything that does pass across the Interface Connection is 8, which is again acceptable. In Figure 83 the security level could have been set at 9, say, which would still have met Rule ID8 & Rule ID9.

In the above, all the examples have discussed maximum security levels. The concepts discussed could also be extended to differentiate between maximum and minimum security level, allowing interfaces to be defined that have a minimum security level of items that they can handle, such as secure networks that can only handle traffic above a given rating.

**Non-Functional Properties**

When defining interfaces it is often useful to include quality of service (QoS) information that details non-functional properties (NFPs) of both the Interface and the operations and items that are invoked and transferred across the Interface. Payne and Fitzgerald define NFPs as those properties that “pertain to characteristics other than functional correctness. For example, reliability, availability and performance of specific functions or services are NFPs that that are quantifiable. Other NFPs may be more difficult to measure: security or adherence to standards, for example.” [Payne & Fitzgerald 2010].
In [Payne&Fitzgerald 2011] they propose extensions to the SysML language to add support for the definition of such NFPs. It is not the intention here to recreate this work which would require a complete SysML profile to be created, but rather to show how the central concepts described by Payne and Fitzgerald could be added to the Interface Pattern through the use of simple stereotyping and SysML’s *trace* mechanisms.

It is worth noting here that the timing extension considered above is really an example of a specific non-functional property (or, perhaps more correctly, a set of related non-functional properties), that of response time. The discussion below on NFPs looks at them from a structural point of view whereas timing was considered above from a behavioural point of view. If a full extension of the Interface Pattern was made to include NFPs (i.e. additions to the concepts in Figure 59, additional views defined on Figure 60 together with Viewpoint definitions and examples) then such an extension would need to include both the structural and behavioural aspects of NFPs. The timing extension would provide an example of one of the behavioural Viewpoints.

A similar case could be made for treating the security extension discussed above as a specific type of non-functional property, although, as discussed in the security extension, the aspects considered are a small subset of those that would perhaps best be covered by a separate security pattern.

Figure 86 shows the definition of non-functional types that will be used in the definition of non-functional properties. Use has been made of SysML’s built-in *value type block*, here stereotyped «nftype» to distinguish from standard usage of *value type blocks*.

![Figure 86 - An example of the definition of non-functional types](image)

Having defined some non-functional types, these can be used to define a number of non-functional properties as shown in Figure 87.
SysML blocks, marked with the «nfproperty» stereotype, have been used in Figure 87 to define two sets of non-functional properties: 'IFProperties' and 'OpProperties'. The intention here is that the 'IFProperties' block defines non-functional properties that will be applied to an Interface Definition, whereas the 'OpProperties' block defines non-functional properties that will be applied to an operation representing the interaction across the Interface defined by the operation's owning Interface Definition. A similar approach could be taken for non-functional properties that apply to Flow Types that are transferred across Flow-Based Interfaces.

If Interfaces or operations have the same sets of properties, then the «nfproperty» blocks can be used by multiple Interfaces and operations. Otherwise each Interface and each operation could potentially have its own associated block defining its non-functional properties. The two diagrams in Figure 86 and Figure 87 could form examples of Views based on additional Viewpoints for the Interface Pattern if desired.

The links between an interface or operation (in SysML terms) and the sets of non-functional properties modelled with «nfproperty» blocks is achieved using SysML’s built-in tracing mechanism.

Having defined sets of non-functional properties and related them to the appropriate element, this information can then be displayed on an extended Interface Definition View as shown in Figure 88.
In the model from which Figure 88 is taken, the 'MsgIF' interface has been traced to the 'IFProperties' block and the 'handleMsg' operation had been traced to the 'OpProperties' block. The call-out comment titled 'Non-functional Properties' attached to 'MsgIF' pulls this information into a single summary comment.

It is important to note that this comment has been automatically generated by the SysML tool used to produce this diagram and that the contents of the comment are automatically updated if the non-functional properties are changed (such as changes to the values, addition of extra properties etc.). This automation and consistency is one of the key drivers for the use of model-based systems engineering.

**Design by Contract**

The definition of Interfaces can be made more formal by using an approach known as “Design by Contract” (DbC). This approach allows the specification of a contract for an Interface by means of pre- and post-conditions defined for the operations of an Interface and invariants over any global properties of an Interface. The pre-condition specifies under what conditions an operation can be executed and the post-condition specifies the condition to expect upon completion if the operation is invoked with the pre-condition satisfied.

Payne & Fitzgerald propose extensions to the SysML language to add support for a DbC approach to Interface definition [Payne&Fitzgerald 2011]. It is not the intention here to recreate this work which would require a complete SysML profile to be created, but rather to show how the central concepts described by Payne and Fitzgerald could be added to the Interface Definition Pattern through the use of simple stereotyping. An example of how this might be done is shown in Figure 89.
Figure 89 shows an Interface Definition View that is a variant of the one presented in Figure 72. Two small changes have been made: the addition of the 'Density' value type to the 'Liquid' block and the addition of the 'Stopped' attribute to the 'DirectionType' enumeration. These have been added for use in the definition of pre- and post-conditions and invariants.

Each of the operations defined for 'PumpIF' have had pre- and post-conditions added. This has been achieved by applying a stereotype to each operation. This stereotype (not shown on the diagram) has two tags associated with it: 'pre' and 'post'. When creating the diagram in Figure 89 these tags have been populated with the pre- and post-conditions for the operation that they are associated with through the applied stereotype. The 'Contract' comment attached to the 'PumpIF' Interface Definition then shows the pre- and post-conditions for each operation of the Interface Definition. It is important to note that this comment has been automatically generated by the SysML tool used to produce this diagram and that the contents of the comment are automatically updated if the pre- or post-conditions are changed. This automation and consistency is one of the key drivers for the use of model-based systems engineering.

A similar approach has been taken with the definition of the invariant shown in the automatically generated 'Invariants' comment. The 'Density' value property of the block used to define the Flow Type has had a stereotype applied (not shown on the diagram) which has an associated tag to hold the details of the invariant. The contents of the attached comment are automatically generated through this stereotype and tag and update automatically if the invariants change.
The pre-conditions defined for ‘FlowIF’ result in a system that behaves differently from that presented earlier and based on the Interface Definition View in Figure 72. In the original version there were no restrictions on the conditions under which the ‘start’, ‘stop’ and ‘reverse’ operations could be invoked. The correct response to these operations was handled through the Protocol Definition View in Figure 80. For example, the PDV handled the receipt of a ‘reverse’ request while the ‘Pump’ was still running. The post-conditions and invariant are consistent with the original definition and could be so applied.

The IDV in Figure 89, at least as far as the pre-conditions are concerned, is intended for a version of the system that does not have an associated Protocol Definition View. For example, the pre-conditions on the ‘start’ and ‘reverse’ operations explicitly state that they can not be invoked unless the Pump is stopped (the ‘CurrentDirection = Stopped’ part of each pre-condition). They also restrict the allowed range of power that can be specified (the ‘0% < powerLevel <= 100%’ part of the pre-conditions); don’t try to start or reverse the pump at zero power or at an overload level (this part of the pre-conditions could be applied to the original definition of the system). The post-conditions on these two operations explicitly state that if the operations are used as specified by their pre-conditions then, upon completion, the pump will be running in the requested direction. The pre- and post-conditions for the ‘stop’ operation are somewhat simpler: a ‘stop’ request may only be requested when the pump is not already stopped and if this is done then the pump will be stopped upon completion of the ‘stop’ operation. The invariant is intended to show that the ‘Density’ of any ‘Liquid’ pumped by the system will never exceed 3000 kg/m³.

The above extension is a light-weight approach to adding design by contract information to the Interface Definition Pattern. For a more detailed approach, see the Contract Pattern in Section 0.

4.4.1.11. Summary

The Interface Definition Pattern provides three Viewpoints that enable the identification and definition of Interfaces to be specified in terms of the structural aspects of the Interfaces: the Interface Identification Viewpoint identifies each Interface, the Interface Connectivity Viewpoint shows the connection between Interfaces and the Interface Definition Viewpoint defines what is transferred across each Interface.

The pattern also provides two Viewpoints that enable the behaviour of Interfaces to be specified: the Interface Behaviour Viewpoint identifies typical scenarios showing how Interfaces are used and the Protocol Definition Viewpoint defines any Protocols to which Interfaces or Ports must conform.

When using the Interface Definition Pattern, as a minimum at least one Interface Connectivity View (ICV) and one Interface Definition View (IDV) are needed to specify Interfaces, their associated Ports and the connections between them. Where the information on the Interface Identification Views (IIVs) is NOT a subset of that on the Interface Connectivity Views, then at least one Interface
Identification View must also be produced. In practice, however, multiple IVs, ICVs and IDVs would be produced along with Interface Behaviour View and, where necessary, Protocol Definition Views. Note here the use of View rather than Viewpoint. When using the Interface Definition Pattern Views are created that conform to the Viewpoints. See [D21.2 2013].

The pattern can also be extended to address issues such as timing, security, non-functional properties (quality of service) and design by contract. Examples of how such extensions might be made were considered and examples given.
4.4.2. Test Pattern

4.4.2.1. Pattern Aims

Testing is an essential part of any system development. Testing must demonstrate two main aims of the development by answering two questions: [Boehm]

- Have we built the right system? (validation)
- Have we built the system right? (verification)

Both verification and validation (V&V) can take on many different forms so it is important that the system can be tested in many different ways, ideally using a single model as a basis for all the testing activities. In the context of safety-critical systems development it is vital to justify that

- Adequate testing activities have been performed, and
- Sufficient test cases have been exercised on the system under test\(^4\).

As a consequence it is desirable to adopt an organised approach to define:
- Test campaigns, that define the creation and execution of test cases
- Test oracles, that compare predicted versus actual results and evaluate the outcome

Such an approach must be produced in a systematic way. If test campaigns and test oracles can exploit tool support automating (some of) the activities involved, then it is essential that these tools conform to such an organised approach, because otherwise unnecessary effort will have to be spent on mapping the artefacts produced by the tools to the evidence required to justify adequateness and comprehensiveness of the test campaign. This organised approach to testing is a model-based approach, known as model-based testing.

*Model-based testing (MBT)* uses specification models from where test cases, test data and expected results, as well as the test procedures executing these cases can be derived. The use of the model will vary enormously and at it two extremes may:

- In its simplest form, the test models specify the expected behaviour of the SUT by identifying interactions of the test environment with the system under test (SUT) in an explicit way, so that these models only represent an abstract view of the test procedures to be exercised on the SUT. The benefits of this simplistic form of modelling are that the model may be used for the automatic generation of executable test cases which will significantly reduce testing time.
- In its most elaborate form, models describe the expected behaviour of the SUT (as above), the execution of the tests (the test campaign) as well as

\(^4\)The most elaborate specification of MBT-related V&V requirements is currently given in the avionic standard [RTCA DO-331].
concrete test data that may be derived algorithmically from the model. Test oracles may also be realised by running the model in back-to-back fashion against the SUT. The benefits of this more-complete form of modelling are myriad and include: reduced testing time, more complete testing, compliance with standards 5.

Clearly, there is more value in the second of these two options, which will require a well-structured and consistent model to realise the full benefits of MBT. This Test Pattern defines a structure of the model that can then be used as a basis for automated model-based testing. The main aims of the Test Pattern are shown in Figure 90.

The diagram in Figure 90 shows that the main aim of the Test Pattern is to ‘Define tests’ which may be broken down into the following:

- ‘Define testing context’. To allow the type of testing, level of testing, testing constraints and any necessary system to be identified and defined by considering the testing context. The depth of testing needs represented by the testing context will dictate which of the benefits of MBT may be realised – simplistic testing needs will result in few benefits, whereas a more comprehensive testing context will result in more benefits
- ‘Define test set-up’. To allow the testing context to be satisfied by defining an overall testing schedule.

5 In [LP10] the authors report efficiency improvements of up to 90% in comparison to the conventional approach, where test cases, test data, and procedures are developed in a manual way.
• ‘Define test cases’. To define the individual test cases that allow all aspects of the system-under-test to be tested.

This pattern can be used for testing any types of systems or system of systems, whether it is software-based or not.

4.4.2.2. Concepts

The main concepts covered by the Test Pattern are shown in the Ontology Definition View (ODV) in Figure 91

Figure 91 Ontology Definition View showing Test Pattern concepts

The diagram in Figure 91 shows the Ontology Definition View for the Test Pattern by identifying the main concepts and the relationships between them. A fundamental part of the Test Pattern is, quite naturally, the ‘Test Case’. One or more ‘Test Case’ is collected into a ‘Test Set’, one or more of which make up a ‘Test Schedule’. However, it is not good enough to simply define a structure for a ‘Test Case’ and then realise it, as we need to understand why the testing is needed in the first place, what are the constraints, what other systems are required, what is the level of testing required and so on. This overall rationale behind the tests is represented by the ‘Testing Context’. The ‘Testing Context’ comprises three main elements:

• One or more ‘Testing Need’. This represents the type of testing that will be carried out (for example: verification, validation, etc.) and will also define any constraints on the testing, such as the scope of the tests, any specific techniques, and so on.
• One or more ‘Required System’. This represents any system that falls outside the testing boundary that may be required for the successful execution of the tests. Examples of such required systems include: the environment, standards, test data, etc.

• ‘Testing Boundary’. The ‘Testing Boundary’ represents the partition between what is included on the model (that satisfies the ‘Testing Need’) to be used for the testing and everything that is not part of the model, but is necessary to perform the testing (the ‘Required System’).

The ‘Test Case’ itself tests one or more ‘Testable Element’. This can be any element from the model of the ‘System Under Test’, any collection of elements or, indeed, the whole model. The main aim here is to simply identify the area of the model that will be tested. This may be visualised by identifying the testable part of the model using a package and then relating the test case to it.

The ‘Test Case’ forms the heart of the pattern and comprises a few key concepts that are important to understand:

• ‘Test Description’, which is quite straightforward and simply defines a text description and a unique identifier for the test case.

• ‘Test Configuration’ defines the configuration of one or more ‘Testable Element’ that is to be tested. This is a structural representation of the ‘Testable Element’ and any other model elements that are connected to it in a specific configuration that is required in order to satisfy a ‘Testing Need’.

• ‘Test Behaviour’ that describes an anticipated behaviour of the ‘Test Case’ that is applied to a ‘Test Configuration’. There will usually be more than one ‘Test Behaviour’ associated with each ‘Test Configuration’ and the anticipated behaviour may represent both desirable and undesirable examples.

• ‘Test Record’. Whereas the ‘Test Behaviour’ describes the test that will be performed by defining the expected behaviour, the ‘Test Record’ captures the actual behaviour that is observed during the execution of the test case.

These concepts are visualised through a number of Viewpoints that form the basis for the Views that form the Test Pattern.

4.4.2.3. Pattern Viewpoints

The Interface Definition pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 92.
The Viewpoint Relationship View in Figure 92 shows that the Test Pattern has three main Viewpoints:

- The 'Testing Context Viewpoint' that defines the rational for the testing activities and identifies a number of needs and constraints that must be satisfied in order that the test campaign be deemed successful.
- The 'Test Set-up Viewpoint' describes the structure of the Test Schedule and Test Set (using the 'Test Structure Viewpoint') and then the execution of the Test Schedule (using the 'Test Schedule Behaviour Viewpoint') and the Test Set (using the 'Test Set Behaviour Viewpoint')
- The 'Test Case Viewpoint' that identifies the scope of the Test Case (using the 'Test Configuration Viewpoint'), the anticipated behaviour of the Test Case (using the 'Test Behaviour Viewpoint') and captures the results of the Test Case (using the 'Test Record Viewpoint')

These Viewpoints are defined in subsequent sections and, for the sake of clarity and readability, will be described in the same way.

4.4.2.4. Rules

Sixteen rules apply to the Test Pattern Viewpoints, as shown in the Rules Definition View (RDV) in Figure 93.
Each Test Set that appears on a Test Structure Viewpoint must appear as a life element for the System Under Test. In this case, the ‘System Under Test’ is the one used in the case study, although one or more other systems that are necessary for the testing activities may also be included.

Figure 93 Rules Definition View showing Test Pattern rules

Note that the sixteen rules shown in Figure 93 are the minimum that are needed. Others could be added if required.

4.4.2.5. Testing Context Viewpoint

The aims of the testing Context Viewpoint are shown in the Viewpoint Context View in Figure 94.

Figure 94 Viewpoint Context View showing Testing Context Viewpoint

The ‘Testing Context Viewpoint’ shown in Figure 94 describes the rationale for the test cases that will be applied to a specific ‘System Under Test’. In this case, the ‘System Under Test’ is the one used in the case study, although one or more...
of the ‘Testing Need’ may be abstracted and re-used on other systems as part of a re-use activity. The ‘Testing Context Viewpoint’ will cover areas such as:

- The reason for the testing, for example: verification or validation
- The level of abstraction of the testing, for example: acceptance testing, systems testing, integration testing, unit testing, etc.
- The type of tests being performed, such: functional testing, performance testing, etc.
- Any other constraints, such as: coverage constraints, success criteria, etc.
- Any external systems that may be required in order to perform the test, such as test rigs, data sets, standards, etc.

It should be stressed that the above list is not intended to be exhaustive but provides some examples of the types of areas that a typical testing context may cover.

The ‘Testing Context Viewpoint’ forms the heart of the Test Pattern and provides the needs that must be satisfied by all the other viewpoints in order that the testing campaign can be deemed to be successful. The diagram in Figure 95 shows the main concepts associated with the Testing Context Viewpoint.

![Figure 95 Viewpoint Definition View showing the elements that appear on the Testing Context Viewpoint](VDV [Package] Testing Context Viewpoint [Test Pattern])

The ‘Testing Context Viewpoint’ will be visualised in SysML using a SysML use case diagram.

The main aim of this viewpoint is to provide the context of the testing activity or, to be more specific, to define:

- The ‘Testing Boundary’ of the testing activity, visualised on a use case diagram using a system boundary
- The ‘Testing Need’ of the testing activity, visualised on a use case diagram using use cases
- Any ‘Required System’ outside the boundary that is necessary to complete the testing activity. This may include a number of elements,
such as the ‘Requirements Model’ that may be necessary for validation, one or more ‘Standard’ that may be required for quality-based testing, specific test data sets, any testing tools needed, etc.

On top of these elements there will also be a number of relationships between the different elements, such as:

- Relationships between the *use cases*, typically the standard SysML dependency-based relations (*include, extend, constrain*, etc)
- Relationships between the *use cases* and the external *actors*, using the standard SysML *associations*.

This viewpoint really drives the whole of the testing activity and will have rules defined that can be applied to ensure consistency within the framework. This viewpoint can also be used to create a library of testing *use cases* and then to show how these *use cases* may be satisfied. This will be particularly useful for specific testing types, for example: equivalence class partition testing may be realised using a particular set of viewpoints (either diagrammatic or mathematical) which may then be used as a basis for generating automated tests.

**Example**

An example of a ‘Testing Context Viewpoint’ can be seen in Figure 96.
The main use case on the diagram in Figure 96 is 'Test system' that includes both 'Verify system' and 'Validate system'. The 'Testing Context View' shown here is quite a comprehensive one that contains aspects of both validation and verification but, of course, the context could be far simpler.

Each of these high-level use cases may be broken down into more detail, for example, the 'Validate system' use case is further decomposed by:

- 'Satisfy all use cases' that ensures that each and every use case from the 'Requirement Model' (note the actor representing the 'Requirement Model') is satisfied.
- 'Satisfy all contexts' that ensure that each and every context from the 'Requirement Model' (note the actor representing the 'Requirement Model') is satisfied.

Other examples could be concerned with defining constraints for the percentage of use cases that must be satisfied. An example of this can be seen in the next diagram.

![Diagram showing constraints on use cases](image)

**Figure 97 An example of an testing Context View defining additional validation constraints**

The diagram in Figure 97 shows how additional constraints may be defined that describe the percentage of use cases that must be satisfied based on their priority.

It is also possible to define the level of abstraction of testing and even some of the test types that need to be followed, as shown in the decomposition of the 'Verify system' use case:

- 'Perform functional testing' and 'Perform QA testing', where the types of tests that need to be applied are defined.
• ‘Apply at all levels’, this constraint shows that the testing must be carried out at various levels of abstraction, in the example here, these levels are: enterprise level, system level, integration level and unit level.

By defining these use cases, it is possible to define the testing context. These use cases will, in part, define the set of ‘Test Case Viewpoint’ as different test cases will lend themselves to particular testing use cases. For example, use cases that refer to validation, system-level testing and integration testing, will lend themselves to scenario-based test cases. This will be discussed in more detail later.

4.4.2.6. Test Set-up Viewpoint

The aims of the Test Set-up Viewpoint are shown in the Viewpoint Context View in Figure 98.

The diagram in Figure 98 shows that the main aims of the ‘Test Set-up Viewpoint’ are to:

• Identify one or more ‘Test Set’ and ‘Test Case’ that make up the ‘Test Schedule’.
• To show how the ‘Test Schedule’ is performed by defining the order of execution of one or more ‘Test Set’.
• To show how each ‘Test Set’ is performed by defining the order of execution of one or more ‘Test Case’.

The Test Set-up Viewpoints do not describe the test themselves but rather the execution of the Test Schedule and its associated Test Sets and Test Cases. The totality of the Test Set-up viewpoints forms the basis of the test campaign. The concepts that are related to the Test Set-up Viewpoint are shown in Figure 99.
The diagram in Figure 100 shows that this viewpoint has three types – the ‘Test Structure Viewpoint’, the ‘Test Schedule Behaviour Viewpoint’ and the ‘Test Set Behaviour Viewpoint’.

The ‘Test Structure Viewpoint’ will be visualised in SysML using a block definition diagram showing:

- The ‘Test Schedule’, represented as a SysML block stereotyped by «Test Schedule», and that is made up of a number of ‘Test Set’.
- One or more ‘Test Set’, represented as SysML blocks that are stereotyped by «Test Set» and that are aggregated into the ‘Test Schedule’.
- One or more ‘Test Case’, represented as SysML operations that are stereotyped by «Test Case».

Dependencies between test sets can also be used to show where one test set may depend on the successful execution of another test set.

The ‘Test Schedule Behaviour Viewpoint’ represents the highest level Viewpoint that describes the order that various ‘Testing Set’ are executed in and the conditions under which they are executed.

The ‘Test Schedule Behaviour Viewpoint’ may be represented in SysML as a sequence diagram where each ‘Test Set’ is visualised using a life line, and the
ordering between them is shown using messages. In order to show the different paths of execution, then combined fragments may be necessary. As each ‘Test Schedule Behaviour Viewpoint’ defines a scenario, it is possible to define a number of these viewpoints (for example showing what happens when all test sets are passed, or when some fail).

The ‘Test Set Behaviour Viewpoint’ shows the order of execution of each ‘Test Case’ that forms part of the ‘Test Set’. This viewpoint will be realised in SysML by a state machine diagram that shows the different states that the ‘Test Set’ may be in and different possible execution paths of each ‘Test Case’.

**Example - Test Structure Viewpoint**
The diagram in Figure 100 shows an example of a ‘Test Structure View’:

The ‘Test Schedule’ in Figure 100 is represented by the block ‘Overall Test Schedule’ (stereotyped as a «Test Schedule»), which represents the testing in its entirety. Clearly, this is a relatively simple case shown here with only a single schedule shown and it is, of course, possible to have many more test schedules defined in this view or, indeed, multiple views.

The ‘Overall Test Schedule’ is made up of a single ‘Validation Test Set’ and one or more ‘Verification Test Set’ both of which are stereotyped as «Test Set».

Each ‘Test Set’ is made up of one or more ‘Test Case’ that are represented here as SysML operations and are stereotyped to «Test Case».
A dependency is also shown here that indicates that the 'Verification Test Set' must be carried out before any instance of 'Validation Test Set' is permitted to be executed.

The 'Test Structure View' shows a structural view of the test schedule and is supported by two behavioural views that represent how the testing is carried out at two different levels of abstraction – these are described in the following two sections.

**Example view - Test Schedule Behaviour View**
The following diagram shows an example of a ‘Test Schedule Behaviour View’ realised by a **sequence diagram**.

![Sequence Diagram for Test Schedule Behaviour View](image)

The 'Test Schedule Behaviour View' in Figure 101 shows the high level execution of the test sets that make up the test schedule. These test sets are taken directly from the ‘Test Structure View’ and must be consistent with that view.

Each of these views may be thought of as representing a testing scenario that satisfies one or more of the testing needs and, therefore, there will typically be a number of these views that represent different testing scenarios and different outcomes.

**Example view - Test Set Behaviour View**
The following diagram shows an example of a ‘Test Set Behaviour View’ realised by a **state machine diagram**.
The ‘Test Set Behavioural View’ shown in Figure 102 will be realised by a SysML state machine diagram that will show the order of execution of the test cases, but may also show the different modes of testing using states. For example, it may be desirable to show ‘normal’, ‘degrading’ and ‘emergency’ modes being tested. The activities that are shown inside the states represent the test cases that are taken directly from the ‘Test Structure View’.

4.4.2.7. Test Case Viewpoint

The aims of the testing Context Viewpoint are shown in the Viewpoint Context View in Figure 103.
The diagram in Figure 103 shows that the main aims of the ‘Test Case Viewpoint’ are to:

- Consider each ‘Test Case’ that was identified in the ‘Test Structure Viewpoint’
- Identify the ‘Element Under Test’ each ‘Test Case’ applies to and to define the configuration of the relevant system elements.
- Define the test behaviours for each ‘Test Case’
- To record the results of applying one or more ‘Test Case’ to the ‘Element Under Test.

The Test Case Viewpoints describe the actual tests themselves, whereas the Test Set-up Viewpoints described the order of execution of the tests.
The diagram in Figure 104 shows that the ‘Test Configuration Viewpoint’ is a structural view that identifies the part of the model that is under test (the ‘Element Under Test’, forming part of the larger ‘System Under Test’) but also identifies any other parts of the model (‘Testable Element’) that may be necessary and, very importantly, the connections between them. This view is more than just a subset of the structural views of the model and must show the relevant connections. This will be visualised in SysML using SysML block definition diagrams and internal block diagrams, where the emphasis will be on the connections between the blocks.

The ‘Test Behaviour View’ will consist of a set of behavioural diagrams that will describe the Test Case that will be executed and its behaviour. This behaviour may represent the desirable behaviour for successful testing or even undesirable behaviour, such as fault or error conditions. The ‘Test Behaviour Viewpoint’ may be visualised using a number of SysML behaviour diagrams:

- **Sequence diagrams.** Used for defining scenarios and high-level testing, such as acceptance testing, integration testing, performance testing, etc.
- **State machine diagrams.** Used for state-based testing, such as unit testing, functional testing, mode-based testing, etc.
• **Activity diagrams.** Used for detailed testing, such as unit testing, functional; testing, process-based testing, etc.

• **Parametric diagrams.** Used for defining mathematical-based scenarios for testing at any level.

The ‘Test Behaviour Viewpoint’ will also form the bridge between the formal-based modelling and the semi-formal modelling, as any of these views may also be realised using formal methods. As illustrated in Appendix A and described in Detail in [D34.1], test cases, test data and test oracles can be automatically derived from formal descriptions of SUT structure and behaviour. To this end, the formal description means of SysML (such as state machines, activities, operations with formal specification of their effect, sequence diagrams, and parametric constraints) may be used to specify a ‘Test Behaviour View’. Alternatively, the view can be described by means of other formal modelling languages such as CSP, Timed Automata or VDM. In any case the criterion for a description to be suitable for test automation is that the expected behavioural semantics\(^6\) of the SUT can be derived from the ‘Test Behaviour Viewpoint’ in a mechanised way.

Each ‘Test Record Viewpoint’ will match its relevant ‘Test Behaviour Viewpoint’ but will represent what actually happens during the execution of the test. Depending on the test approach (defined in the ‘Testing Context’) it may be appropriate to only record unexpected test results or it may be necessary to record results for a given ‘Test Case’, both expected and unexpected. This view may be realised in a number of ways:

• In simple cases, it may be appropriate to simply mark up a copy of the original ‘Test Behaviour Viewpoint’, but then these mark-ups must be referenced in the model using notes.

• Creating a new view, using the same type of SysML diagram as the original ‘Test Behaviour Viewpoint’, but recording what actually happened, rather than what was expected to happen.

The ‘Test Record Viewpoint’ will not be considered in great detail here, as it uses the same visualisation as the ‘Test Behaviour Viewpoint’.

**Example**

\(^6\) The behavioural semantics of a model or a system consists of the set of all computations it can perform. A computation is an infinite sequence of states transitions. Events and other communication concepts can also be encoded in the state space, so every possible behavior is expressible by such a computation. A requirement is reflected in the model by a subset of its computations. Test cases verifying a requirement are traces, that is, finite prefixes of some computations reflecting the requirement. Therefore automated tracing from requirements to test cases, test results and vice versa is possible, if the elements of the ‘Test Behaviour View’ are linked to the requirements.
The ‘Test Configuration View’ will be visualised using a SysML *block definition diagram* but the visualisation of the ‘Test Behaviour View’ will vary depending on the type of test, level of abstraction, etc.

In order to show these different visualisations, a number of examples will be considered and both the ‘Test Configuration View’ and the ‘Test Behaviour View’ will be shown for each. For each of these different examples, the testing need will be shown by relating the test case views back to the original ‘Testing Context’ that forms part of the pattern. It should be stressed here that the examples provided are intended to give an indication of the types of testing that can be carried out and this is not intended to be an exhaustive list.

**Validation using stakeholder scenarios**

The stakeholder scenario represents a high level of abstraction of testing. In the example considered here, taken from a model of an escapology stunt, the interactions between the stakeholders and the stunt will be explored in the form of a scenario in order to validate some of the original *use cases* for the stunt.

![Figure 105 Example 'Test Configuration View' showing the stakeholders](image)

The ‘Test Configuration View’ shown in Figure 105 is realised by a *block definition diagram* that shows the stakeholder for the stunt as *blocks*. The stakeholder-level scenarios focus on the interactions between the stakeholders and the system when considered as a ‘black box’.

A number of scenarios may now be considered to satisfy the original *use cases* for the stunt.
The 'Test Behaviour View' in Figure 106 shows a scenario that represents the desired correct performance of the stunt, where everything goes according to plan.

The 'Test Case View' shown in this example is used for validation of the use cases in the original requirements model for the stunt. Bearing on mind that the purpose behind this testing activity is validation, then by using this example of the 'Test Case View' it is possible to satisfy one of the use cases from 'Testing Context View', which satisfies the rule 'Each 'Test Case' must satisfy at least one testing need, represented as a use case in the testing context'.

The partial diagram in Figure 107 shows the use case that this 'Test Case View' is satisfying, in particular, 'Satisfy all use cases'.
Verification at the system level using systems scenarios

The system scenario represents a high level of abstraction of testing. In the example considered here, the processes behind the stunt will be executed in the form of a scenario in order to verify the stunt at a system level.

The 'Test Configuration View' shown in Figure 108 is realised by a block definition diagram that shows the processes as blocks and the activities that are performed during each process are shown by operations.

Based on this structural view, it is possible to consider a number of different scenarios using sequence diagrams.

The 'Test Behaviour View' shown in Figure 109 is realised by a sequence diagram that shows the execution of the processes for a specific use case from the original stunt requirements model. The scenario here represents typical successful performance of the stunt.
The 'Test Behaviour View' shown here is also realised using a sequence diagram, but this time the view illustrates a different scenario that must be considered in order to satisfy the original use case for the stunt. The scenario here shows one of the scenarios where the stunt goes wrong and emergency action is taken.

The 'Test Case View' shown in the example is used for verification of the original stunt. Bearing in mind that the purpose behind this testing activity is verification at a system level, then by using this example of the 'Test Case View' it is possible to satisfy one of the use cases from 'Testing Context View', which satisfies the rule 'Each ‘Test Case’ must satisfy at least one testing need, represented as a use case in the testing context'.

The 'Test Case View' shown in Figure 111 may be considered to be used for performing functional testing, for the purposes of verification, at the system level.
Verification at the integration level using system scenarios

The previous example was focussed on the processes that are executed to verify the stunt at the system level, whereas the 'Test Case View' here is focussed on verification, but this time at the integration level.

The 'Test Configuration View' shown in Figure 112 is visualised by a block definition diagram that focusses on the element that make up the physical part of the system and the interfaces between them.

The different ways that these elements interact may be explored using systems scenarios that are visualised using sequence diagrams.

Figure 112 Example 'Test Configuration View' for system

Figure 113 Example 'Test Behaviour View' for system-level scenario for normal operation
The ‘Test Behaviour View’ in Figure 113 shows how the system elements interact for a scenario with two possible outcomes – when the ‘Emergency’ flag is set to true or false.

The ‘Test Case View’ shown in this example is used for verification of the original stunt. Bearing in mind that the purpose behind this testing activity is verification at the integration level, then by using this example of the ‘Test Case View’ it is possible to satisfy one of the use cases from ‘Testing Context View’, which satisfies the rule ‘Each Test Case must satisfy at least one testing need, represented as a use case in the testing context’.

The ‘Test Case View’ shown in Figure 114 may be considered to be used for performing functional testing, for the purposes of verification, at the integration level.

**Verification at the unit level using system element-based test**

The ‘Test Case View’ considered in this section will be applied for verification purposes at the system element level. The previous two verification-related examples applied at the system and integration levels, whereas this example looks at an individual element of the physical system.

```
<table>
<thead>
<tr>
<th>«block»</th>
<th>Pump</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>prime ()</td>
<td></td>
</tr>
<tr>
<td>flush ()</td>
<td></td>
</tr>
<tr>
<td>pump ()</td>
<td></td>
</tr>
<tr>
<td>pumpReverse ()</td>
<td></td>
</tr>
<tr>
<td>stopPump ()</td>
<td></td>
</tr>
<tr>
<td>ports</td>
<td></td>
</tr>
<tr>
<td>out pOut : FluidFlow</td>
<td></td>
</tr>
<tr>
<td>in  pIn : ~FluidFlow</td>
<td></td>
</tr>
<tr>
<td>values</td>
<td></td>
</tr>
<tr>
<td>Rate : m3/s</td>
<td></td>
</tr>
<tr>
<td>CurrentDirection : PumpDirection</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 115 Example 'Test Configuration View' for single system element
The ‘Test Configuration View’ in Figure 115 is visualised by a block definition diagram that shows a single system element represented as a block. The emphasis here is on showing the properties, operations and ports of the block.

![Diagram showing a block definition diagram for a single system element.](image)

**Figure 116 Example ‘Test Behaviour View’ for a single system element**

The ‘Test Behaviour View’ shown in Figure 116 uses a state machine diagram to show the desired internal operation of the system element that was identified in the ‘Test Configuration View’. This view may be used as a basis for functional testing where all the possible paths in the state machine are explored. It is also possible to show different state machines for the same system element in order to show, for example, different modes of operation.

The ‘Test Case View’ shown in this example is used for verification of the original stunt. Bearing in mind that the purpose behind this testing activity is verification at the unit level, then by using this example of the ‘Test Case View’ it is possible to satisfy one of the use cases from ‘Testing Context View’, which satisfies the rule ‘Each ‘Test Case’ must satisfy at least one testing need, represented as a use case in the testing context’.

![Diagram showing the flow of use cases from 'Testing Context View' to 'Test Case View'.](image)

**Figure 117 Satisfying use cases form the 'Testing Context View'**
The ‘Test Case View’ shown in Figure 117 may be considered to be used for performing functional testing, for the purposes of verification, at the unit level.

**Verification at the unit level using system process-based test**

The previous example considered unit-level testing focussing on a single system element. It is also possible to consider a single process at the unit level and to test its operation.

The ‘Test Configuration View’ shown in Figure 118 shows a simple *block definition diagram* where a single process is represented as a *block*, with its *activities* shown as *operations*.

The ‘Test Behaviour View’ in Figure 119 shows the internal operation of a single process (identified in the ‘Test Configuration View’) using an *activity diagram*.
The ‘Test Case View’ shown in this example is used for verification of the original stunt. Bearing in mind that the purpose behind this testing activity is verification at the unit level, then by using this example of the ‘Test Case View’ it is possible to satisfy one of the use cases from ‘Testing Context View’, which satisfies the rule ‘Each Test Case must satisfy at least one testing need, represented as a use case in the testing context’.

Verification of quality assurance processes

The previous example showed how processes that are executed as part of the system may be tested. This section looks how processes that form part of the quality system of the organisation may be tested.

In this example, a testing process exists that tests the coverage of the test cases, in particular the original needs of the system are satisfied by at least one scenario. This test process may itself be satisfying a requirement of a particular standard.

The ‘Test Configuration View’ in Figure 121 is visualised by a block definition diagram that shows ontological elements and the relationships between them. These relationships represent the traceability paths between the elements that are used as a basis for the coverage testing.
The ‘Test Behaviour View’ in Figure 122 shows the internal operation of the testing process (identified in the ‘Test Configuration View’) using an activity diagram.

The ‘Test Case View’ shown in this example is used for verification of the original stunt. Bearing in mind that the purpose behind this testing activity to perform QA testing according to a standard, then by using this example of the ‘Test Case View’ it is possible to satisfy one of the use cases from ‘Testing Context View’, which satisfies the rule ‘Each ‘Test Case’ must satisfy at least one testing need, represented as a use case in the testing context’.

Figure 122 Example ‘Test Behaviour View’ executing a testing process

Figure 123 Satisfying use cases form the ‘Testing Context View’
The ‘Test Case View’ shown in Figure 123 may be considered to be used for performing QA testing, for the purposes of verification.

**Parametric-based scenario at multiple levels**
The examples provided so far have focussed on testing at different levels of abstraction, but some aspects of testing may be applied at many different levels. When considering parametric-based scenarios, it is possible to apply them to any aspect of the system model that can be measured in some way.

The example here will focus on showing parametric-based testing at the integration level, but the same principles may be applied at any level of abstraction, for both validation and verification.

![Diagram of system model](image)

**Figure 124 Example 'Test Configuration View' for parametric values**

The ‘Test Configuration View’ in Figure 124 uses a block definition diagram to show the main system elements and the relationships between them. In particular, the properties of each block are shown as these will be used as a basis for the measurement used for the parametrics.
The ‘Test Behaviour View’ in Figure 125 is visualised by a *parametric diagram* that show how different equations and heuristics may be applied to the system properties to explore a number of different scenarios. Depending on the whether property values are: constants, variables or fixed variables, it is possible to change their values, where permitted, to perform trade-off analysis and optimisation.

The ‘Test Case View’ shown in this example of applying parametric-based testing to the system. Parametric-based testing may be potentially used to satisfy any one of the original *use cases* from the ‘Testing Context View’.

### 4.4.2.8. Summary

This section has presented an overview of the Test Case Pattern. The pattern consists of three core Viewpoints.

The Testing Context Viewpoint drives the entire testing activity and identifies a number of testing use cases that must be satisfied by the combined testing activity described in all the other test Views. This view is visualised using a *use case diagram*.

The Test Set-up Viewpoint identifies the test schedule, test sets and test cases (in the Test Structure Viewpoint, a type of Test Set-up Viewpoint, visualised by a *block definition diagram*) and then describes their behaviour at the test schedule level (the Test Schedule Behaviour Viewpoint, another type of Test Set-up Viewpoint, visualised using a *sequence diagram*) and the test set level (the Test Set Behaviour Viewpoint, the third type of Test Set-up Viewpoint, visualised using a *state machine diagram*).

The Test Case Viewpoint identifies the element under test and its associated system elements (through the Test Configuration Viewpoint, a type of Test Case Viewpoint, visualised using a block definition diagram) and the test behaviour itself (the Test Behaviour Viewpoint, another type of Test Case Viewpoint, visualised using a number of different diagrams). The results of the test are
recorded in the Test Record Viewpoint (the third type of Test Case Viewpoint) that will take the form of a marked-up copy of the Test Behaviour Viewpoint or an equivalent View showing what actually happened during the test.

The most important aspect of this pattern is to consider the rationale of the testing activities using the Testing Context Viewpoint and to ensure that all the other Viewpoints contribute to satisfying the *use case* described here.
4.4.3. Traceability Pattern

Background
There are many interpretations of traceability, including:

- Measurement
- Logistics
- Materials
- Supply Chain
- Development

However, two themes run throughout all interpretations: an unbroken chain and access to information. The unbroken chain focuses on the need to be able to follow an artefact back to its source, whether that source is a physical place such as the sea, a supplier organisation, a requirement, meeting minutes or a standard such as ISO 15288. Access to information defines the need to know about the artefact being traced. In some cases this may be the history of the artefact. For example, in the case of food, this could be the organisations the food has passed through, when and who did the quality check etc. For an engineering project, it may be why an artefact exists or has been chosen as the solution to a requirement.

The existence of traceability supports areas of analysis including: impact analysis, change management and coverage analysis. All of these types of analysis would be difficult if not impossible to carry out without traceability in place.

A note on the use of the term ‘System Element’ in this pattern: This pattern uses the term ‘System Element’ throughout, defining a pattern for modelling interfaces between system elements. This pattern can be used for modelling interfaces at both the system and system-of-systems level. When used at the system level, then a system element will be a part of the system (for example, a sub-system or component or assembly, depending on the system hierarchy being employed). When used at the system-of-systems level, then a system element will be a constituent system of the SoS under consideration. Put simply an SoS is a system whose system elements are constituent systems.

4.4.3.1. Pattern Aims

This pattern is intended to be used as an aid to the use of traceability within a Model-Based Systems Engineering (MBSE) application. The main aims of this pattern are shown in the Architectural Framework Context View (AFCV) in Figure 126.
The key aim of the Traceability Pattern is to allow a ‘Systems Engineer’ to ‘Establish traceability’ in a model of a System. This key aim is constrained by the need to:

- ‘Support traceability throughout model’ – Support the capture of the traceability relationships between traceable elements in a systems engineering model, throughout the model and between any desired model elements.
- ‘Support impact analysis’ – Support ‘Systems Engineers’ and ‘Systems Engineering Managers’ in the identification of model elements that may be impacted by change.

The main aim of ‘Establish traceability’ includes the aim of ‘Define allowed traceability’. This in turn includes the uses cases:

- ‘Define types of trace allowed’ – Support the definition of the types of traces that can be used.
- ‘Define types of element that can be traced to/from’ – Support the definition of the types of elements that can be involved in trace relationships and the relationships that can be used between such traceable elements.

### 4.4.3.2. Concepts

The main concepts covered by the Traceability Pattern are shown in the Ontology Definition View (ODV) in Figure 127.
Key to getting traceability right is establishing the understanding of the way in which traceability will be used. This means understanding the types of information to be traced and the relationships that can be used to realise the traceability.

The right-hand side of this diagram shows the types of things which can be traced - a 'Traceable Type'. These may be

- 'Viewpoint', representing types of 'View' occurring in a model. 'Viewpoint' may represent underlying diagram types from the modelling language being used, such as a *block definition diagram* if using SysML or a *class diagram* if using UML. They may also represent defined Viewpoints from a framework that is being used, such as a "Validation Viewpoint" from a Model-Based Requirements Engineering framework. Viewpoints are conceptual in nature; they are a definition to which an instance (a View) must conform. See [Holt & Perry 2013].

- 'Viewpoint Element', representing types of 'View Element' occurring in a model. Viewpoint Elements may represent underlying modelling element types from the modelling language being used, such as a *block* if using SysML or a *class* if using UML. They may also represent defined conceptual elements (Ontology Elements) that are used on Viewpoints from a framework that is being used, such as a "System Element" that is used on a "Validation Viewpoint" from a Model-Based Requirements Engineering framework. Viewpoint Elements are conceptual in nature and make up a Viewpoint; when an instance of a Viewpoint is created (i.e. a View is created), then that View is made up of View Elements. See [Holt & Perry 2013].
The right-hand side of the diagram also includes ‘Relationship Type’ that define the types of ‘Traceability Relationship’ that can be used to trace View Elements and Views, identified by the Viewpoints and Viewpoint Elements, one to another. The left-hand side of the diagram shows ‘Traceable Element’. This represents the actual things being traced between i.e. the ‘View’ or ‘View Element’. The traceability is made using a ‘Traceability Relationship’, a representation of the actual relationship which is being considered.

The concepts on the right-hand side of the diagram are conceptual. They represent the types of things that can be traced (‘Traceable Type’) which can be a ‘Viewpoint’ or a ‘Viewpoint Element’, together with the type of relationships that can be used (‘Relationship Type’). The concepts on the left-hand side of the diagram are concrete. They represent the actual things traced (‘Traceable Element’) which can be a ‘View’ or a ‘View Element’, together with the type of traceability connecting them (‘Traceability Relationship’).

Many people consider and define traceability; fewer fully consider the way in which traceability is defined. Often where it is defined it is considered as a database schema, which although useful may not provide a direct relationship to the way a project is being conducted or what that project is trying to achieve though the provision of traceability.

### 4.4.3.3. Pattern Viewpoints

The Traceability Pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 128.

![VRV Framework - Traceability](image-url)
The Traceability Pattern defines four Viewpoints to enable the definition and capture of traceability:

- **The ‘Relationship Identification Viewpoint’** is used to define the types of permissible Relationship Types.
- **The ‘Traceability Identification Viewpoint’** is used to define the items that can be traced (the Traceable Types) and the types of trace that can be used between items (the Relationship Types between pairs of Traceable Types).
- **The ‘Traceability Viewpoint’** is used to capture and visualise traceability between Traceable elements through Traceability Relationships. This may be in the form of a diagram, table, matrix etc.
- **The ‘Impact Viewpoint’** is used to show a traceability tree for a selected Traceable Element from a Traceability View, allowing the items potentially impacted by changes to the root of the tree (the selected Traceable Element) to be identified.
### Rules

Five rules apply to the four Traceability Viewpoints, as shown in the Rules Definition View (RDV) in Figure 129.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule TR1</th>
<th>Rule TR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule Text</td>
<td>All Traceability Relationships must be defined as Relationship Types on a Relationship Identification Viewpoint.</td>
<td>All Traceability Relationships must be defined as Relationship Types on a Relationship Identification Viewpoint.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule TR3</th>
<th>Rule TR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule Text</td>
<td>As a minimum one Traceability Identification Viewpoint, one Relationship Identification Viewpoint and one Traceability Viewpoint must be produced.</td>
<td>The permitted Relationship Types that can occur between each pair of Traceable Types must be defined on a Traceability Identification Viewpoint.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule TR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule Text</td>
<td>If a Traceable Type, T1, has a defined Relationship Type, R1, to another Traceable Type, T2, then every Traceable Element defined by Traceable Type T1 must have a corresponding Traceability Relationship of type R1 to another Traceable Element defined by Traceable Type T2.</td>
</tr>
</tbody>
</table>

Note that the five rules shown in Figure 129 are the minimum that are needed. Others could be added if required.
4.4.3.5. Relationship Identification Viewpoint (RIV)

The aims of the Relationship Identification Viewpoint are shown in the Viewpoint Context View in Figure 130.

The main aim of the Relationship Identification Viewpoint is to ‘Establish traceability’ through the aim of ‘Define allowed traceability’. In particular, its key aim is to ‘Define types of traces allowed’, that is the Relationship Identification Viewpoint identifies the allowed Relationship Types that can be used to establish traceability.

**Description**

The Viewpoint Definition View (VDV) in Figure 131 shows the Ontology Elements that appear on a Relationship Identification Viewpoint.
The Relationship Identification Viewpoint shows a number of Relationship Types.

Relationship Type defines the relationships that will be used to capture traceability. In many cases, traceability relationships are assumed to be loose relationships added by a requirements or traceability tool. However, this Viewpoint also enables other concepts such as parent/child relationships to represent traceability.

A Relationship Identification Viewpoint would typically be based on the Ontology of a specified pattern or application (such as requirements engineering), adding detail relating to the types of traceability relationship to fully define the information to be captured.

Example
An example View that conforms to the Relationship Identification Viewpoint is shown in Figure 132. This, and subsequent examples, are taken from the domain of Model-Based Systems Engineering (and Model-Based Requirements Engineering in particular). However, this is only one possible application of traceability and, while perhaps the commonest use within the systems engineering domain and hence the reason it was chosen for the examples, it is not the only one.

The Relationship Identification View in Figure 132, realised as a SysML block definition diagram, defines nine Relationship Types: ‘Trace’, ‘Refinement’, ‘Derivation’ etc., that have been identified as being the kinds of traceability needed when carrying out model-based requirements engineering.

Each of the defined Relationship Types should also be accompanied by a description of its intended use. This has been done in the following table, which also serves as a text-based version of a Relationship Identification View:
### Relationship Identification View

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>A general purpose relationship type that can be used if none of the other, more specific, relationship types is suitable. For example, to indicate that a requirement traces to a source element.</td>
</tr>
<tr>
<td>Refinement</td>
<td>Indicates that one requirement refines another OR that a use case refines a requirement.</td>
</tr>
<tr>
<td>Derivation</td>
<td>Indicates that one requirement is derived, in whole or part, from another requirement.</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Indicates that a requirement satisfies a rationale or capability OR that a capability satisfies a goal OR that a system element satisfies a use case.</td>
</tr>
<tr>
<td>Verification</td>
<td>Indicates that a test case verifies a requirement.</td>
</tr>
<tr>
<td>Validation</td>
<td>Indicates that a test case validates a use case.</td>
</tr>
<tr>
<td>Constraint</td>
<td>Indicates that one requirement constrains another OR that one use case constrains another.</td>
</tr>
<tr>
<td>Inclusion</td>
<td>Indicates that one requirement includes another as a subrequirement OR that one use case includes another.</td>
</tr>
<tr>
<td>Extension</td>
<td>Indicates that one use case extends another under stated circumstances.</td>
</tr>
</tbody>
</table>

Table 4 - An example of a text-based Relationship Identification View for MBRE

This View, defined in Figure 132 and Table 4, fulfils Rule TR2. With the Traceability Identification View in Figure 135 and Table 5 and the Traceability View in Figure 138, it fulfils Rule TR1.

### 4.4.3.6. Traceability Identification Viewpoint (TIV)

The aims of the Traceability Identification Viewpoint are shown in the Viewpoint Context View in Figure 133.
The main aim of the Traceability Identification Viewpoint is to 'Establish traceability' through the aim of 'Define allowed traceability'. In particular, its key aim is to 'Define types of elements that can be traced to/from', that is the Traceability Identification Viewpoint identifies the allowed Traceable Types as well as the allowed Relationship Types that can be used to establish traceability between any two Traceable Types.

**Description**

The Viewpoint Definition View (VDV) in Figure 134 shows the Ontology Elements that appear on the Traceability Identification Viewpoint.

![Figure 134 - Viewpoint Definition View showing the elements that appear on the Traceability Identification View (TIV)](image)

The Traceability Identification Viewpoint identifies the types of elements that can be involved in traceability relationships, along with the types of trace that can be used between them.

The Traceability Identification Viewpoint shows a number of Traceable Types and Relationship Types.

Traceable Types are defined as Viewpoints and Viewpoint Elements. These define the model viewpoints and elements that can be the source and targets of traceability. If a type of viewpoint or element is not identified on a Traceability Identification View then no trace relationship will be able to be defined from or to that element or viewpoint.

The Relationship Types (defined on the Relationship Identification Viewpoint) are used on the Traceability Identification Viewpoint to define the allowed relationships between pairs of Traceable Types. Only those Relationship Types defined between a pair of Traceable Types can be used between them.
The Traceability Identification Viewpoint will be based on the Ontology of a specified pattern or application (such as requirements engineering), adding detail relating to the types of items that are involved in traceability, along with the nature and direction of the traceability relationships between them to fully define the information to be captured.

**Example**
An example View that conforms to the Traceability Identification Viewpoint is shown in Table 5.

<table>
<thead>
<tr>
<th>Traceability Identification View</th>
<th>Traceable Type (Viewpoint or Viewpoint Element)</th>
<th>Relationship Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>Requirement</td>
<td>Trace</td>
</tr>
<tr>
<td></td>
<td>Source Element Requirement</td>
<td>Derivation</td>
</tr>
<tr>
<td></td>
<td>Requirement</td>
<td>Inclusion</td>
</tr>
<tr>
<td></td>
<td>Rationale</td>
<td>Constraint</td>
</tr>
<tr>
<td></td>
<td>Capability</td>
<td>Refinement</td>
</tr>
<tr>
<td></td>
<td>Use Case Requirement</td>
<td>Rationale</td>
</tr>
<tr>
<td></td>
<td>Use Case</td>
<td>Satisfaction</td>
</tr>
<tr>
<td></td>
<td>System Element Requirement</td>
<td>Satisfaction</td>
</tr>
<tr>
<td></td>
<td>Test Case (Validation View)</td>
<td>Satisfaction</td>
</tr>
<tr>
<td></td>
<td>Test Case</td>
<td>Validation</td>
</tr>
<tr>
<td></td>
<td>Requirement Context View</td>
<td>Stakeholder Role (on Context Definition Viewpoint)</td>
</tr>
<tr>
<td></td>
<td>Capability</td>
<td>Goal</td>
</tr>
<tr>
<td></td>
<td>Use Case</td>
<td>Satisfaction</td>
</tr>
</tbody>
</table>

Table 5 - An example of a text-based Traceability Identification View for MBRE

The Traceability Identification View shown in Table 5 defines the Traceable Types (Viewpoints and Viewpoint Elements) and Relationship Types that can exist between them for a Model-Based Requirements Engineering (MBRE) application.

The first two columns define the Traceable Types, giving the source and destination of possible relationships. The final column defines the possible relationships that can hold between each pair of Traceable Types. Only those Relationship Types defined on a Relationship Identification View may appear in this column.

The same kind of information as shown in Table 5 may also be shown graphically as in Figure 135, which shows a subset of the information found in Table 5.
This SysML block definition diagram shows an extract from a Model-Based Requirements Engineering framework. In this diagram there are three Viewpoints defined, the ‘Context Definition Viewpoint’, ‘Requirement Context Viewpoint’ and the ‘Validation Viewpoint’. These are marked as such with the «Viewpoint» stereotype. A number of Viewpoint Elements are also shown, again indicated as such through the use of stereotypes. Note that for the Viewpoint Elements ‘Requirement’ and ‘Source Element’, their owning Viewpoints are not shown (simply to reduce the complexity of this example diagram).

The blocks marked as representing Viewpoints and Viewpoint Elements indicate those elements which can be traced. The associations with the «Trace» stereotype indicate the possible traceability Relationship Types that are valid between the blocks linked by the association. Where the «Trace» stereotype has been applied, further information about the detail of the trace is also shown through the use of tags:

- ‘Trace to’ - the direction of the trace
- ‘Trace type’ - the type of trace, e.g. Trace, Validate, Inclusion etc.

One item of note on the diagram concerns the association between the ‘Stakeholder Role’ (from the ‘Context Definition Viewpoint’) and the ‘Requirement Context Viewpoint’. The trace indicated on this association is shown as applying in the opposite direction to the reading direction of the association. The association is read from ‘Stakeholder Role’ to ‘Requirement Context Viewpoint’ but the trace goes to the ‘Stakeholder Role’ rather than from it. This is due to the nature of traceability, which is often considered as going back towards the source of the information. The intention here is to show that a ‘Requirement Context Viewpoint’ can be traced back to its source ‘Stakeholder Role’ on a ‘Context Definition Viewpoint’.

Figure 135 - An Example of a Traceability Identification View for MBRE
This View, defined in Figure 135 and Table 5, fulfils Rules TR3 and TR4. With the Relationship Identification View in Figure 132 and Table 4 and the Traceability View in Figure 138, it fulfils Rule TR1.

4.4.3.7. Traceability Viewpoint (TV)

The aims of the Traceability Viewpoint are shown in the Viewpoint Context View in Figure 136.

The main aim of the Traceability Viewpoint is to ‘Establish traceability’ in a model, with the constraint of ‘Support traceability throughout model’. That is, the Traceability Viewpoint allows traceability to be captured and visualised throughout the model of a System; it is not something that is restricted to Requirements traceability. Traceability Views that conform to the Traceability Viewpoint may be in the form of diagrams, tables, matrices, trees, etc.

**Description**

The Viewpoint Definition View (VDV) in Figure 137 shows the Ontology Elements that appear on a Traceability Viewpoint.
The Traceability Viewpoint shows a number of Traceable Elements and the Traceability Relationships between them. Traceable Elements may be Views or View Elements and Traceability Relationships may be made View-to-View, View Element-to-View Element, View Element-to-View or View-to-View Element.

The allowed Traceability Relationships are those of the Relationship Types that are defined on a Relationship Identification View. The allowed Traceable Elements are those of the Traceable Types that are defined on a Traceability Identification View. Only the connections between pairs of Traceable Types defined on a Traceability Identification View are permitted on a Traceability View.

**Example**
An example View that conforms to the Traceability Viewpoint is shown in Figure 138.
The Traceability View in Figure 138, here realised as a SysML requirements diagram, shows six different types of Traceability Relationship between a number of Traceable Elements:

- Two Inclusion relationships between the sub-Requirements ‘Computer-controlled Pump’ and ‘Allow Different Fluids’ and their parent Requirement ‘Perform Stunt’, realised using SysML nesting relationships.
- A Derivation relationship between the Requirement ‘Provide Pump Controller’ and the Requirement ‘Computer-controlled Pump’ that it is derived from, realised using a SysML «deriveReqt» relationship.
- Two Refinement relationships between the Use Cases ‘Perform using concrete’ and ‘Perform using custard’ and the Requirement ‘Allow
Different Fluids’ that they refine, realised using a SysML «refine» relationship.

- Two Validation relationships between the Test Cases ['Package] Scenarios [Stunt Performance with Medium-density Concrete]' and ['Package] Scenarios [Stunt Performance with Custard]' and the Use Cases 'Perform using concrete' and 'Perform using custard' that they validate, realised using a SysML dependency stereotyped «validate». Note that SysML does not have a "built-in" «validate» relationship.

As an alternative to a graphical representation, the same information could be represented as a simple matrix as shown in Table 6.

| Traces From | \( \text{requirement}\) Perform Stunt | \( \text{requirement}\) Computer-controlled Pump | \( \text{requirement}\) Allow Different Fluids | \( \text{requirement}\) Provide Pump Controller | Test Cases [Computer Control of Pump - Successful Stunt] | Use Cases Perform using concrete | Use Cases Perform using custard |
|-------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|

Table 6 – An example of a Traceability View represented as a matrix

Note that the Traceability View defined in Figure 138 and Table 6 has been created to show the traceability, both forwards and backwards, for a single Requirement, ‘Perform Stunt’. However, this is only one possibility. There is no restriction on the focus that each Traceability View has nor on the number of levels shown. A Traceability View could be created showing, for example, a number of Source Elements and all the Requirements that trace to them (at a single level of traceability), or could concentrate on a single Source Element and everything that ultimately traces to this (multi-level traceability); this would be similar to Figure 138 but would show multiple Requirements tracing to the Source Element.
This View, defined in Figure 138 and Table 6, along with the Relationship Identification View in Figure 132 and Table 4 and the Traceability Identification View in Figure 135 and Table 5, fulfils Rules TR1, TR2, TR3 and TR4.

4.4.3.8. Impact Viewpoint (IV)

The aims of the Traceability Viewpoint are shown in the Viewpoint Context View in Figure 139.

![Figure 139 - Viewpoint Context View showing Impact Viewpoint aims](image)

The main aim of the Impact Viewpoint is to ‘Establish traceability’ in a model, with the constraints of ‘Support traceability throughout model’ and ‘Support impact analysis’. An Impact View that conforms to the Impact Viewpoint is used to show a traceability tree for a selected Traceable Element from a Traceability View, allowing the items (the other Traceable Elements connected to it through a network of Traceability Relationships) that may be impacted by changes to the root of the tree (the selected Traceable Element) to be identified.

**Description**

The Viewpoint Definition View (VDV) in Figure 140 shows the Ontology Elements that appear on an Impact Viewpoint.
Like the Traceability Viewpoint, the Impact Viewpoint shows a number of Traceable Elements and the Traceability Relationships between them. Indeed, the Ontology Elements that can appear on the Impact Viewpoint and the rules governing consistency of those elements are the same as for the Traceability Viewpoint.

Where the two views differ is in the form and intent of the Views created based on the Viewpoints. A Traceability View can show any number of levels of traceability and may or may not have a single "root" element that everything traces to or from. An Impact View, however, always has a single root element and typically shows all the levels of traceability starting from that root element. Whereas a Traceability View may show a web of related elements, an Impact View always shows a tree of related elements, starting from the element that is the root of the tree.

**Example**
An example View that conforms to the Impact Viewpoint is shown in Figure 141.
The Impact View in Figure 141 shows the forward impact tree for the Source Element ‘Initial Ideas Meeting 10.01.2008’, showing all fifteen potential model elements that may be impacted by a change to the Source Element. Note that there is an implicit meaning in the layout of the diagram: items in a given level trace to the item lying over them in the level above. The nature of the trace relationship is shown in the stereotype that forms the first part of the name of each item and the type of item traced is shown in the stereotype that forms the last part of the name. For example, from the diagram it can be seen that the Use Cases ‘Build stunt coffin’ and ‘Ensure coffin not crushed by fluid’ both trace, through refinement, to the Requirement ‘Crush-proof Coffin’.

Other graphical representations are, of course, possible. The same information may also be presented as text, as shown in Table 7. Here, the level of indentation is used to show the hierarchical relationships between the elements; those at the same level trace to the element above them that is indented one level less.
Whatever the graphical representation used, what is essential is that such Impact Views are capable of being generated automatically based on the trace information added to the model of a System. The information in Figure 141 and Table 7 was generated automatically from the SysML tool that was used to create the model.

4.4.3.9. Summary

The Traceability Pattern defines four Viewpoints that allow aspects of traceability to be captured. The Relationship Identification Viewpoint defines the possible types of traceability relationships that may be used. The Traceability Identification Viewpoint defines the types of model elements that can be the sources and targets of traceability and the relationships (from the Relationship Identification Viewpoint) that can hold between them. The Traceability Viewpoint shows the actual traceability relationships that hold between model elements.
elements that conform to the allowed types for traceability (as defined on the Traceability Identification Viewpoint). Finally, the Impact Viewpoint allows a traceability impact tree to be produced for a selected root model element, aiding in the identification of those model elements that may be impacted by changes to the selected root element.
4.4.4. Life Cycle Pattern

Background

All systems that are developed pass through a number of life cycles. The nature of the various life cycle that a system goes through are often misunderstood. There are many different types of life cycle that exist, including, but not limited to:

- **Project Life Cycles.** Project Life Cycles are perhaps, along with Product Life Cycles, one of the most obvious examples of applications of Life Cycles. We tend to have rigid definitions of the terminal conditions of a Project, such as start and end dates, time scales, budgets and resources and so a Project Life Cycle is one that many people will be able to identify with.

- **Product Life Cycles.** Another commonly considered life cycle is that of the product (system) itself. It is relatively simple to visualise the conception, development, production, use, support and disposal of a product.

- **Programme Life Cycles.** Most projects will exist in some sort of higher-level programme. Each of the programmes will also have its own life cycle which will have some constraints on the life cycles of all projects that are contained within it.

- **System Procurement Life Cycles.** Some systems may have a procurement life cycle that applies to them. From a business point of view, this may be a better way to view a product, or set of products, than looking at the Product Life Cycle alone.

- **Technology Life Cycles.** Any technology will have a life cycle. For example, in the past, the accepted norm for removable storage was magnetic tapes. This was then succeeded by magnetic discs, then optical discs, then solid-state devices and then virtual storage. Each of these technologies has its own life cycle.

- **Equipment Life Cycles.** Each and every piece of equipment will have its own life cycle. This may start before the equipment is actually acquired and may end when the equipment has been safely retired. Stages of the equipment life cycle may describe its current condition, such as whether the equipment is in use, working well, degrading and so on.

- **Business Life Cycle.** The business that acquires or engineers or project manages or uses a system will have a life cycle. In some cases, the main driver of the business may be to remain in business for several years and then to sell it on. Stages of the life cycle may include expansion or growth, steady states, controlled degradation and so on.

These different types of life cycle not only exist, but often interact in complex ways. Also, each of these life cycles will control the way that processes are executed during each stage in the life cycle. The Life Cycle Pattern is intended to allow the definition, behaviour and interaction of the various life cycles of a system to be defined and understood.
4.4.4.1. Pattern Aims

This pattern is intended to be used as an aid to the understanding of the life cycles involved in the MBSE development of a system. The main aims of this pattern are shown in the Architectural Framework Context View (AFCV) in Figure 142.

![AFCV Diagram]

Figure 142 - Architectural Framework Context View showing Life Cycle Pattern aims

The key aim of the Life Cycle Pattern is to allow a ‘Systems Engineer’ and a ‘Systems Engineering Manager’ to ‘Understand system life cycles’. This key aim is constrained by the need to ‘Apply to any type of life cycle’, including (but not limited to):

- Project life cycles (‘Apply to project life cycles’)
- Product life cycles (‘Apply to product life cycles’)
- Procurement life cycles (‘Apply to procurement life cycles’)

The main need to ‘Understand system life cycles’ includes the need to understand a life cycle in term of its structure (‘Understand lifecycle structure’) and its behaviour (‘Understand life cycle behaviour’). Both of these include the need to ‘Understand interactions between life cycles’, allowing multiple Life Cycles for a System to be related one to another.

4.4.4.2. Concepts

The main concepts covered by the Life Cycle Pattern are shown in the Ontology Definition View (ODV) in Figure 143.
Figure 143 – Ontology Definition View showing Life Cycle concepts

The main concept is that of a ‘Life Cycle’, a set of one or more ‘Stage’ that can be used to describe the evolution of system, project etc. over time. Each ‘Stage’ represents a period within a ‘Life Cycle’ that relates to its realisation through one or more ‘Process Execution Group’ (see discussion below). The success of a ‘Stage’ is assessed by a ‘Gate’, a mechanism for assessing the success or failure of the execution of a ‘Stage’. These Ontology Elements allow the structural aspects of a Life Cycle to be captured.

The behavioural aspects are represented by the concept of a ‘Life Cycle Model’ which represents the execution of a set of one or more ‘Stage’, showing the behaviour of a ‘Life Cycle’.

It is important to understand the difference between a Life Cycle and a Life Cycle Model:

- A Life Cycle is a structural construct that shows the Stages that make up a Life Cycle.
- A Life Cycle Model is a behavioural construct that shows the Stages behave within the execution of a Life Cycle.

A ‘Life Cycle’ may interface with others. This is done through a ‘Life Cycle Interaction Point’. Each corresponding ‘Life Cycle Model’ will then interact with others through a ‘Life Cycle Interaction’, the point during a ‘Life Cycle Model’ at which they interact, reflected in the way that one or more ‘Stage’ interact with each other.

The concept of the ‘Process Execution Group, an ordered execution of one or more process that is performed as part of a ‘Stage’, is a linking concept that can be used to relate the above Life Cycle concepts to those of process modelling and project planning. This is discussed further in the section on the Life Cycle Model Viewpoint below. For a full discussion see [Holt & Perry 2013].
4.4.4.3. Pattern Viewpoints

The Life Cycle Pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 144.

![Diagram of Viewpoint Relationship View showing Life Cycle Viewpoints](image)

The Life Cycle pattern defines four Viewpoints:

- The ‘Life Cycle Viewpoint’ that identifies the Stages that exists in a Life Cycle.
- The ‘Life Cycle Model Viewpoint’ that describes how each Stage behaves over time in relation to one or more other Stage.
- The ‘Interaction Identification Viewpoint’ that identifies the Life Cycle Interaction Points between one or more Life Cycle.
- The ‘Interaction Behaviour Viewpoint’ that describes the execution of each Life Cycle Interaction Point in relation to one or more other Life Cycle Interaction Point as identified in an Interaction Identification Viewpoint.

Each of these Viewpoints is described in more detail in the following sections. For each Viewpoint an example View that conforms to the Viewpoint is also given.

4.4.4.4. Rules

Five rules apply to the four Pattern Viewpoints, as shown in the Rules Definition View (RDV) in Figure 145.
Note that the five rules shown in Figure 129 are the minimum that are needed. Others could be added if required.

4.4.4.5. Life Cycle Viewpoint (LCV)

The aims of the Life Cycle Viewpoint are shown in the Viewpoint Context View in Figure 146.

The main aim of the Life Cycle Viewpoint is to ‘Understand system life cycles’ in terms of their structure (‘Understand life cycle structure’), in a way that can ‘Apply to any type of life cycle’.

Description
The Viewpoint Definition View (VDV) in Figure 147 shows the Ontology Elements that appear on a Life Cycle Viewpoint.

The Life Cycle Viewpoint is a relatively simple Viewpoint, showing the structure of a single Life Cycle in terms of the Stages that it is made up of.

**Example**
An example View that conforms to the Life Cycle Viewpoint is shown in Figure 148.

This View fulfils Rule LC1.

4.4.4.6. Life Cycle Model Viewpoint (LCMV)

The aims of the Life Cycle Model Viewpoint are shown in the Viewpoint Context View in Figure 149.

The main aim of the Life Cycle Model Viewpoint is to ‘Understand system life cycles’ in terms of their behaviour (‘Understand life cycle behaviour’), in a way that can ‘Apply to any type of life cycle’.

**Description**

The Viewpoint Definition View (VDV) in Figure 150 shows the Ontology Elements that appear on a Life Cycle Model Viewpoint.
The Life Cycle Model Viewpoint shows the Stages and Gates that make up a single Life Cycle Model, showing the behaviour of a Life Cycle through time and the Gates controlling behaviour at the end of each Stage.

As noted in the discussion on the Ontology for the Life Cycle Pattern, the concept of the Process Execution Group, an ordered execution of a number of processes that is performed as part of a Stage, is a linking concept that can be used to expand a Life Cycle Model to tie the concepts of process modelling to those of project planning. It allows the Life Cycle Pattern, through the Life Cycle Model Viewpoint, to bridge the gap often found in systems engineering between the defined systems engineering processes followed by the systems engineers developing a system and the project plans produced by project managers tasked with managing the development of the system. This is done by expanding the definition of the Life Cycle Model Viewpoint to allow Views based on the Viewpoint to show the Process Execution Groups that are executed in a particular Stage, along with the sequencing and timing of such executions. Further expansion allows the processes within a Process Execution Group to be shown in a similar fashion. Such changes would require additions to the Ontology and changes to the VDV for the Life Cycle Model Viewpoint. For a full discussion of this, see [Holt & Perry 2013].

*Example*
An example View that conforms to the Life Cycle Model Viewpoint is shown in Figure 151.
The Life Cycle Model View in Figure 151, realised as a SysML sequence diagram, shows a typical development Life Cycle Model based on the Life Cycle defined in Figure 148. Each Stage is represented by a life line and corresponds to one of the Stages represented using blocks on Figure 148. Each life line also shows an execution specification (the small rectangles) that may be annotated to include timing constraints or parallelism between various Stages. The Gate at the end of the 'Conception' Stage is shown as an internal message with the stereotype «gate».

If the expanded Life Cycle Model Viewpoint discussed above is being used, then additional Views would be produced, one for each Stage, showing the Process Execution Groups taking place within each Stage. A further set would typically also be produced, one for each Process Execution Group, showing the processes executed with each Process Execution Group. See [Holt & Perry 2013] for a full discussion.

The Life Cycle Model View in Figure 151 along with the Life Cycle View in Figure 148, fulfils Rule LC2 and Rule LC4.

**4.4.4.7. Interaction Identification Viewpoint (IIV)**

The aims of the Interaction Identification Viewpoint are shown in the Viewpoint Context View in Figure 152.
The main aim of the Interaction Identification Viewpoint is to ‘Understand system life cycles’ in terms of their interactions (‘Understand interactions between life cycles’) from a structural point of view (‘Understand life cycle structure’), in a way that can ‘Apply to any type of life cycle’.

**Description**

The Viewpoint Definition View (VDV) in Figure 153 shows the Ontology Elements that appear on a Interaction Identification Viewpoint.
The Interaction Identification Viewpoint shows one or more Life Cycles, the Stages that make them up and the Life Cycle Interaction Points between the Stages. Note that this shows structural information; it shows which Stages from which Life Cycles have interactions but not when they interact.

**Example**

An example View that conforms to the Interaction Identification Viewpoint is shown in Figure 154.

![Figure 154 - An example of an Interaction Identification View](image)

The Interaction Identification View in Figure 154, realised as a SysML block definition diagram, shows three Life Cycles, the ‘Development Life Cycle’, ‘Acquisition Life Cycle’ and ‘Deployment Life Cycle’. Each of these Life Cycles is represented using a package with the stereotype «life cycle». The Stages in each Life Cycle are realised using blocks with the stereotype «stage». Each Life Cycle Interaction Point is represented as a dependency with the stereotype «interaction point». Thus, for example, there is a Life Cycle Interaction Point between the ‘Delivery’ Stage of the ‘Acquisition Life Cycle’ and the ‘Concept’ Stage of the ‘Development Life Cycle’, represented by the «interaction point» dependency between them. Note that the full term Life Cycle Interaction Point is not used here for the stereotype on the dependencies purely for reasons of clarity and presentation.

The Interaction Identification View in Figure 154, along with the Life Cycle View in Figure 148, fulfils Rule LC3.

**4.4.4.8. Interaction Behaviour Viewpoint (IBV)**

The aims of the Interaction Behaviour Viewpoint are shown in the Viewpoint Context View in Figure 155.
The main aim of the Interaction Behaviour Viewpoint is to ‘Understand system life cycles’ in terms of their interactions (‘Understand interactions between life cycles’) from a behavioural point of view (‘Understand life cycle behaviour’), in a way that can ‘Apply to any type of life cycle’.

**Description**

The Viewpoint Definition View (VDV) in Figure 156 shows the Ontology Elements that appear on an Interaction Behaviour Viewpoint.

The Interaction Behaviour Viewpoint shows one or more Life Cycle Models, the Stages that make them up and the Life Cycle Interactions between the Life Cycles.
These Life Cycle Interactions must correspond to the Life Cycle Interaction Points between the Stages as defined on the corresponding Interaction Identification View. Note that the Interaction Behaviour View shows behavioural information; it shows how the Life Cycles interact through time.

**Example**

Two example Views that conform to the Interaction Behaviour Viewpoint are shown in Figure 157 and Figure 158.

![Figure 157 - An example of an Interaction Behaviour View](image)

Figure 157 shows an Interaction Behaviour View realised as a SysML *sequence diagram*. It shows the Life Cycle Interactions for a single Life Cycle (the ‘Development Life Cycle’). The Stages of the Life Cycle are represented by the *life lines*. Each Life Cycle Interaction is represented by a *message* with the stereotype «interaction» which enter and exit the *sequence diagram* via *gates*. (Note: these are SysML *gates*, part of the notation of the *sequence diagram*. They are NOT the same as the Ontology Element ‘Gate’ that appears on Figure 143). Note also that the full term Life Cycle Interaction is not used here for the stereotype on the *messages* purely for reasons of clarity and presentation.

With the visualisation used in Figure 157 it should be noted that the other Life Cycles that are interacted with are not shown, simply the Life Cycle Interaction Points entering and leaving via *gates*. If the interactions between the different Life Cycle Models are particularly important, then the alternate visualisation shown in Figure 158 may be considered.
The diagram in Figure 158 has a similar visualisation to that shown in Figure 157 but this time the Life Cycle Interaction Points do not enter and exit the diagram anonymously, but interact with specific Life Cycles Models. Each Life Cycle Model is shown as a package with each Stage visualised as a life line. This time, however, the Life Cycle Interaction Points do not end in a gate but go to other Stages (the life lines) within other Life Cycle Models (the packages).

The Interaction Behaviour Views in Figure 157 and Figure 158, along with the Interaction Identification View in Figure 154, fulfil Rule LC5.

### 4.4.4.9. Extensions to the Life Cycle Pattern

The Life Cycle Model Viewpoint presented here is the minimal version of the Viewpoint; it can be extended to show the Process Execution Groups that are executed in each Stage of a Life Cycle Model as well as the processes executed within each Process Execution Group. This allows the use of the Life Cycle Pattern to bridge the gap often found in systems engineering between the defined systems engineering processes followed by the systems engineers developing a system and the project plans produced by project managers tasked with managing the development of the system. For a full discussion of this, see [Holt & Perry 2013].

### 4.4.4.10. Summary

The Life Cycle Pattern defines four Viewpoints that allow the aspects of a system’s Life Cycle to be captured. The Life Cycle Viewpoint identifies the Stages that exists in a Life Cycle. The Life Cycle Model Viewpoint describes how each Stage behaves over time in relation to the other Stages. The Interaction Identification Viewpoint identifies the Life Cycle Interaction Points between a number of Life Cycles. Finally, the Interaction Behaviour Viewpoint describes the execution of each Life Cycle Interaction Point in relation to other Life Cycle Interaction Points as identified in an Interaction Identification Viewpoint.
4.4.5. Epoch Pattern

**Background**

The model lies at the heart of all the systems engineering activities in the world of model-based systems engineering. There is a need, therefore, to manage and control the evolution of the model over time. In order to manage and control a model, it is essential to be able to measure the model and reason about those measurements. This is the driving force behind the Epoch Pattern. The main aim of this pattern is to identify a specific point in time where the System Model is baselined and then to reason about that model. These specific points in time are defined as Epochs and the reasoning is enabled by Measures and Metrics.

4.4.5.1. Pattern Aims

This pattern is intended to be used as an aid to the definition of epochs. The main aims of this pattern are shown in the Architectural Framework Context View (AFCV) in Figure 159.

![Figure 159 - Architectural Framework Context View showing Epoch aims](image)

The diagram in Figure 159 shows the context for the Epoch Pattern. The main aim of the Epoch Pattern is to understand the evolution of a system (‘Understand evolution’) by considering the evolution of the complexity of the system over time (‘Support evolution of system through time’).

There is a single high-level constraint that makes an assumption that a model of the System must exist (‘Ensure model exists’). All of the Measure and Metrics that are defined using the Epoch Pattern Views are applied to the System model, therefore if it does not exist then no Measure nor Metrics may be applied.
There are three main use cases that must be satisfied for the Epoch Pattern to be effective which are:

- ‘Define epoch’, where a point in time is identified and an Epoch is defined at that point in time.
- 'Identify key viewpoints', where the System Model is looked at and a number of Viewpoints are identified that have been deemed relevant for the Epoch ('Identify relevant viewpoints'). Alongside the Viewpoints, it is also important that the Context of these Viewpoints is also known ('Identify contexts').
- 'Measure model', which includes defining Measure and Metrics ('Define general Metrics') and also applying them to the System Model ('Apply Metrics'). The results of the application of the Metrics must also be presented to the modeller ('Present results') and then the model must be allowed to be updated or augmented based on these results ('Augment model'). There is a constraint that is applied to the ‘Measure model’ use case which is that automation of the application of the Metrics must be possible ('Ensure automation is possible').

This context drives the Epoch Pattern.

4.4.5.2. Concepts

The main concepts covered by the Epoch Pattern are shown in the Ontology Definition View (ODV) in Figure 160.

![ODV Diagram](image-url)

Figure 160 – Ontology Definition View showing Epoch concepts

The Ontology for the Epoch pattern is shown in Figure 160 and describes the concepts and terminology, as follows:
• ‘Epoch’ that defines a point during the Life Cycle of the Project or System that will form a reference point for Project activities. An Epoch will apply to a specific baseline of the System Model, but not all baselines will be an Epoch. Epochs may be evolved from other Epochs, and may evolve into other Epochs.

• ‘Applicable Viewset’ that is a subset of the ‘System Model’ and comprises one or more ‘View’. These Views that make up the Epoch will be visualised by one or more ‘Diagram’ (in the case of the COMPASS Project, these will be SysML diagrams).

• ‘System Model’ is the abstraction of a specific System that evolves during the Life Cycle of the Project or System.

• ‘View’ is a defined set of information (based on a Viewpoint) with a standard structure that may be realised with one or more ‘Diagram’. The Views make up the ‘System Model’.

• ‘Diagram’ that is a visualisation of a ‘View’ and that uses a particular notation, in this case the language will be SysML.

• ‘Node’ is any graphical node (typically a shape) that forms part of the modelling language that is used in a Diagram.

• ‘Path’ is any graphical path (typically a line) that forms part of the modelling language that is used in a Diagram.

• ‘Measure’ that is an operation performed on a ‘Diagram’ that yields a result, such as a counting of blocks on a block definition diagram, counting interactions on a sequence diagram, etc.

• ‘Metric’ that uses one or more ‘Measure’ in order to provide meaningful knowledge about a System Model, for example a Metric may calculate coupling, cohesion, etc.

The elements defined on Ontology Definition View form the basis for all of the Viewpoints that are defined as part of the Epoch Pattern.

4.4.5.3. Pattern Viewpoints

The Epoch Pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 161.
The Epoch Pattern defines four Viewpoints that are described as follows:

- The ‘Epoch Definition Viewpoint’ that identifies and describes the Epoch that is being defined.
- The ‘Applicable Viewset Viewpoint’ that identifies the subset of the System Model that is relevant to the Epoch being defined.
- The ‘Metric Definition Viewpoint’ where all of the Metrics and Measures are defined.
- The ‘Metric Usage Viewpoint’ that shows how the Metrics and Measures (defined in the Metric Definition Views) are applied to the Applicable Viewset (defined in the Applicable Viewset Views).

Each of these Viewpoints is described in more detail in the following sections. For each Viewpoint an example is also given.

4.4.5.4. Rules

Five Rules apply to the four Epoch Viewpoints, as shown in the Rules Definition View (RDV) in Figure 162:
Note that the five Rules shown in Figure 162 are the minimum that are needed, and are described at a high level as follows:

- ‘Rule EP1’ - ‘The System Model must exist’. This Rule enforces the fundamental assumption that a System Model must exist as all of the Measure and Metrics that are used as part of the Epoch Pattern are applied to the System Model. Therefore, if there is no System Model, then the Measure and Metrics cannot be applied.

- ‘Rule EP2’ - ‘The System Model must have a pre-defined set of Viewpoints that form the basis for the Applicable Viewset’. This Rule follows on from EP1 as, not only must a model exist, but it must be structured into a set of Viewpoints that describe its Views. It is these Viewpoints and Views that are used to identify the Applicable Viewset.

- ‘Rule EP3’ - ‘All Measures and Metrics that are defined must relate to the Diagrams that are used to visualise Views in the Applicable Viewset’. This Rule follows on from EP1 and EP2 and states that the Measures and Metrics that are defined must be applied to Diagrams that exist in the Applicable Viewset.

- ‘Rule EP4’ - ‘The value for each parameter on each Measure must be taken directly from a Diagram, Node or Path’. This Rule follows on from EP3 and enforces the fact that each parameter for a Measure must be taken directly from a Diagram, Node or Path. When combined with Rule EP3, it is ensured that the Measures are valid according to the Diagrams, Nodes or Paths, and that these themselves are taken directly from the Applicable Viewset.

- ‘Rule EP5’ - ‘The value for each parameter on each Metric must be taken directly from a Measure, Diagram, Node or Path’. This Rule is similar to EP4 except here it is the Metrics that are being enforced. In the case of Metrics, their parameters must be taken directly from a Diagram, Node or Path, but may also be taken from a Measure.
This is the minimum set of Rules. Others could be added if required.

4.4.5.5. Epoch Definition Viewpoint (EDV)

The aims of the Epoch Definition Viewpoint are shown in the Viewpoint Context View in Figure 163.

The diagram in Figure 163 shows that the main aim of the Epoch Definition Viewpoint is to ‘Define epoch’. This is broken down further by the following three Use Cases:

- ‘Identify epoch’, where a specific point during the Life Cycle of the Project or System is identified.
- ‘Define epoch characteristics’, where the properties associated with the Epoch are defined, such as ‘Date’, ‘Version’ and ‘Model ref’.
- ‘Define relationships to other epochs’, which entails looking at other epochs in the Life Cycle and identifying relationship between them and the new epoch. This will allow the evolution of the model to be explored.

It should be stressed that the aim of this Viewpoint is to define the Epochs that show the evolution of the System Model over time, but not to define exactly what aspect of the System Model is being measured (such as size, complexity, etc.). The elements that are relevant for the Epoch Definition Viewpoint are shown in the next section.

Description

The Viewpoint Definition View (VDV) in Figure 164 shows the Ontology Elements that appear on an Epoch Definition Viewpoint.
The diagram in Figure 164 shows the elements from the Ontology that are relevant to the Epoch Definition Viewpoint. The Epoch Definition Viewpoint shows the Epoch being defined with its property values filled in and the relationships between them.

**Example**
An example View that conforms to the Epoch Definition Viewpoint is shown in Figure 165.
The Epoch Definition View in Figure 165, realised as a *block definition diagram*, shows two *instance specifications* of Epoch, ‘Epoch 1’ and ‘Epoch 2’ that are connected together by the ‘evolves from’ link.

The Epochs shown on this View effectively show the evolution of a System Model over time. The aspects of this evolution will depend on what is being measured. For example, in the example used in this section, it is the complexity of the model that is being measured. Therefore, it is the evolution of the complexity of the System Model that will be shown in this example.

Any number of Epochs may be shown using this View, ranging from a single Epoch to a string of Epochs that are sequenced together. It should also be noted that the Epochs need not be connected in a linear fashion; it is also possible to have branches to show divergence in the evolution of the System Model.

### 4.4.5.6. Applicable Viewset Viewpoint (AVV)

The aims of the Applicable Viewset Viewpoint are shown in the Viewpoint Context View in Figure 166.

The diagram in Figure 166 shows that the main aim of the Applicable Viewset Viewpoint is to 'Identify key viewpoints', which has three types associated with it:

- 'Identify relevant viewpoint', that requires the subset of the System Model that is relevant for the Epoch being defined to be identified.
- 'Identify context', where the Context refers to the Context of the Viewpoints that have been identified (hence the constraint between the two use cases). This is important as it is possible that the Context for those Viewpoints has evolved as time has gone on and, hence, this evolution must be captured.
- ‘Identify key views’, where these Views will be the specific instances of the Viewpoints that have been identified.

There is also a constraint, ‘Limit to existing views’ that really enforces rules ‘Rule EP1’ and ‘Rule EP2’ from Figure 162.

**Description**
The Viewpoint Definition View (VDV) in Figure 167 shows the Ontology Elements that appear on an Applicable Viewset Viewpoint.

![Diagram](VDV Package Applicable Viewset Viewpoint Epoch Pattern)

**Figure 167 – Viewpoint Definition View showing the elements that appear on the Applicable Viewset Viewpoint (AVV)**

The Applicable Viewset Viewpoint shows how the ‘Applicable Viewset’ is related to a specific ‘Epoch’ and comprises a one or more ‘View’ and one or more ‘Viewpoint’.

**Example**
An example View that conforms to the Applicable Viewset Viewpoint is shown in Figure 168.
The Applicable Viewset View in Figure 64, realised as a SysML block definition diagram, shows the Viewpoints that are relevant to a specific Epoch. In this example, the Viewpoints are the ones that are defined by the Interface Pattern.

Note how the Interface Pattern is being re-used here to define which Viewpoints are relevant to this Applicable Viewset View. As well as knowing which Viewpoints form part of the Applicable Viewset View, it is essential that the actual Views are also identified. This is because it is possible and likely that neither the entire System Model, nor an entire set of instances of a specific Viewpoint will be relevant.

These Views are shown on the diagram as notes that are associated with their relevant Viewpoint and that provide a reference for each View that is relevant for each Viewpoint.

4.4.5.7. Metric Definition Viewpoint (MDV)

The aims of the Metric Definition Viewpoint are shown in the Viewpoint Context View in Figure 169.
The diagram in Figure 169 shows that the main aim of the Metric Definition Viewpoint is to 'Measure model', which includes 'Define general Metrics' that has four types:

- ‘...for coupling’ that is a generalisation relating to complexity. This requires that general Metrics are defined that measure and calculate some aspect of the complexity between Nodes.
- ‘...for cohesion’ that is a generalisation relating to complexity. This requires that general Metrics are defined that measure and calculate some aspect of the complexity inside a Node.
- ‘... for structural diagrams’ that is generalisation related to an aspect of the System Model. This requires that the general Metrics can be applied to the structural aspect of the System Model.
- ‘... for behavioural diagrams’ that is generalisation related to an aspect of the System Model. This requires that the general Metrics can be applied to the behavioural aspect of the System Model.

There is a general constraint that requires that any Metrics developed must be able to be automated (‘Ensure automation is possible’).

**Description**

The Viewpoint Definition View (VDV) in Figure 170 shows the Ontology Elements that appear on a Metric Definition Viewpoint.
The Metric Definition Viewpoint shows the Measures and the Metrics associated with them. Each Measure and Metric is defined here in terms of its Equations and associated Equation Definitions and Parameters.

**Example**

An example View that conforms to the Metric Definition Viewpoint is shown in Figure 171.
The Metric Definition View in Figure 171 realised as a SysML block definition diagram. This View is used to show Measures and Metrics, both of which are visualised using SysML constraint blocks. The constraint block has three compartments that are used as follows:

- **Constraint block name** – this is the name of the Measure or Metric. Also, stereotypes are used to show whether the constraint block is representing a Metric («metric») or a Measure («measure»).
- ‘constraints’ compartment, where the equation that represents the Measure or Metric is defined.
- ‘parameters’ compartment, where the parameters that are used by the constraints are defined according to their type and multiplicity.

The diagram in Figure 171 shows four Metrics that are relevant to the Epoch used in this example.

These Metrics are defined as follows.

- ‘Coupling’, that calculates the coupling between different nodes on a single diagram, and that uses the ‘Average IO Metric’ [CRU80]. The equation for this Metric is:
  - \( \text{Coupling} = \frac{\text{SUMOF}(Z_i...Z_n)}{n} \), where:
    - \( Z \) is the average number of inputs and outputs for the node \( i \)
    - \( n \) is the total number of nodes.
- $i$ is the count of the nodes on the diagram
- $n$ is the total number of nodes on the diagram

- ‘Average IO Metric’, that calculates the average number of inputs and outputs shared over $m$ nodes [CRU80]. The equation for the Metric is:
  - $Z = \frac{\text{SUMOF}(M_j...M_n)}{n}$, where:
    - $M$ is the sum of the inputs and outputs shared by nodes $j$ and $i$
    - $M$ is the total number of nodes to which $i$ is joined

- 'Structural Cohesion Metric', that calculates the average number of operations per block in a diagram [LOR94]. The equation for the Metric is:
  - $SC = \frac{\text{SUMOF}(O_i...O_n)}{n}$, where:
    - $O$ is the number of operations in a single block
    - $n$ is the number of blocks

- 'Behavioural Cohesion Metric', that calculates the complexity of a behavioural diagram by calculating the possible number of paths through its decision structure [MCC76]. The equation for the Metric is:
  - $V = e - n + p$, where:
    - $e$ is the number of nodes
    - $N$ is the number of paths
    - $P$ is the number of start and end points

The Metrics that are defined here are used purely as an example of how established techniques may be applied to the Applicable Viewset for a specific Epoch. These Metrics were traditionally applied at the code level in software engineering in the original references, but have been tailored to be able to be applied at the model level – see [TUG01] for more information.

### 4.4.5.8. Metric Usage Viewpoint (MUV)

The aims of the Metric Usage Viewpoint are shown in the Viewpoint Context View in Figure 172.
The diagram in Figure 172 shows that the main aim of the Metric Usage Viewpoint is to 'Measure model', which includes being able to apply the Metrics to the System Model ('Apply Metrics').

There is a general constraint that requires that any Metrics developed must be able to be automated ('Ensure automation is possible').

**Description**

The Viewpoint Definition View (VDV) in Figure 173 shows the Ontology Elements that appear on a Metric Usage Viewpoint.
The Metric Usage Viewpoint in Figure 173 shows how the Measures and Metrics are applied to the System Model via the Diagrams.

**Example**

Example Views that conform to the Metric Usage Viewpoint are shown in Figure 174.

<table>
<thead>
<tr>
<th>Measure/Metric</th>
<th>Applied to</th>
</tr>
</thead>
</table>

The Metric Usage View in Figure 174 is realised as simple table. The table relates the Metrics and Measures that were defined in the Metrics Definition View to the relevant Views in the Applicable Viewset View.

Typically, in SysML, the usage of constraints is shown using a parametric diagram. In this case, however, the Metrics are being applied to the meta-level of the System Model rather than directly to the System Model itself. For example, when counting the number of operations on a block, it is the number of operations that appear at the meta-level, rather than the operation itself that is being measured.

**4.4.5.9. Summary**

This section has introduced and defined the set of Viewpoints that are used to realise a set of Epoch Views that make up the Epoch Pattern.

The Epoch Pattern allows modellers to define a set of Measures and Metrics that may then be applied to a Model, or sub set of a Model.
It should be noted that the Epoch Pattern provides a mechanism that allows modellers to define Measures and Metrics, but the pattern itself does not define any Metrics or Measures. The pattern itself is purely definitional in nature. An example set if Metrics is shown here based on previous work on applying Metrics and Measures to Models,
4.4.6. Contract Pattern

Background
A key challenge in SoS engineering is in the analysis of, and maintaining, global emergent properties under SoS evolution and integration. Whilst the Interface Pattern allows an engineer to define the data and interactions between CSs, it is insufficient in modelling the internal behaviours of those CSs as is required to understand emergence. As such, the Contract Pattern is intended to allow the engineer to specify limited internal behaviours to which CSs are required to conform to order for them to a part of the SoS. CSs may conceivably conform to multiple contracts (and not necessarily contracts of the SoS of interest), and may conform to the contract in any way they wish.

Note that the Contract Pattern should be used in combination with the Interface Pattern.

4.4.6.1. Pattern Aims

It is intended that this pattern be used as an aid to the definition of contracts. The main aims of this pattern are shown in the Architectural Framework Context View (AFCV) in Figure 175.

The key aim of the Contract Pattern is to ‘enable the analysis of SoS emergent behaviour’, which requires the use case to ‘model the SoS in terms of contracts’. This use case includes further use cases for modelling the SoS contracts and identifying conformance to those contracts:
- 'Identify contracts in an SoS' – where the SoS engineer determines the various contracts required to provide the emergent behaviours of the SoS, which must restrict the behaviours acceptable for a CS to join the SoS.
- 'Identify conformance of constituent systems to contracts' – to ensure that the CSs of a SoS conform to contracts defined by the SoS engineer and so that the engineer is able to dictate the conformance relations in the SoS.
- 'Identify connections between contracts' – enables the SoS engineer to identify how the contracted behaviours should be linked in an SoS.
- 'Define contract functionality' – defining the contracts in terms of the state and operations they provide to their environment and internal operations.
- 'Define ordering of external communications and internal state' – definition of any protocols to which a contract must conform.

4.4.6.2. Concepts

The main concepts covered by the Contract Pattern are shown in the Ontology Definition View (ODV) in Figure 176.

![Ontology Definition View showing Contract Pattern concepts](image)

The key concepts of the pattern are the ‘Contractual SoS’ and ‘Contract’. A ‘SoS’ is composed of two or more ‘Constituent Systems’ – as is well understood in the literature and in the COMPASS project. A ‘SoS’ conforms to several ‘Contractual
SoSs’, which is composed of two or more ‘Contracts’. A ‘Constituent System’ conforms to several ‘Contracts’. A ‘Contract’ may be composed of several ‘Contracts’.

Both ‘Constituent System’ and ‘Contract’ elements own ‘Ports’ which expose ‘Interfaces’. This relationship is consistent with the Interface Pattern and it is recommended that the interfaces are defined using the Interface Pattern. The ontologies of the two patterns are consistent in this respect and thus are compatible.

A ‘Contract’ has; ‘State Variables’, ‘State Invariants’ which constrain the ‘State Variables’, ‘Operations’ which may modify and read the ‘State Variables’ and ‘Protocols’ which control how the contract behaves in terms of its ‘Operations’.

### 4.4.6.3. Pattern Viewpoints

The Contract Pattern defines a number of Viewpoints as shown in the Viewpoint Relationship View (VRV) in Figure 177.

![Figure 177 - Viewpoint Relationship View showing Contract Pattern viewpoints](image)

The Contract Pattern defines five Viewpoints for the definition of Contracts:

- The ‘Contractual SoS Definition Viewpoint’ is used for the identification of Contracts comprising the Contractual SoS.
- The ‘Contract Connections Viewpoint’ is used to show how Contracts are connected through their interfaces.
The ‘Contract Definition Viewpoint’ is used for the definition of Contracts in terms of their state variables, state invariants and operations they may provide.

The ‘Contract Conformance Viewpoint’ is used for defining the conformance relationship between CSs and Contracts.

The ‘Contract Protocol Definition Viewpoint’ is used for the definition of any Protocols to which a Contract must conform.

Each of these Viewpoints is described in more detail in the following sections. For each Viewpoint an example is also given.

### 4.4.6.4. Rules

Seven rules apply to the five Contract Pattern Viewpoints, as shown in the Rules Definition View (RDV) in Figure 178:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP1</td>
<td>Every Contract on a Contract Conformance Viewpoint must be identified on a Contractual SoS Definition Viewpoint.</td>
</tr>
<tr>
<td>CP2</td>
<td>All Contracts on a Contractual SoS Definition Viewpoint must have an associated Contract Definition Viewpoint.</td>
</tr>
<tr>
<td>CP3</td>
<td>At a minimum, a Contractual SoS Definition Viewpoint, a Contract Connections Viewpoint, a Contract Definition Viewpoint and a Contract Protocol Definition Viewpoint are required.</td>
</tr>
<tr>
<td>CP4</td>
<td>All Interfaces present in a Contract Connections Viewpoint must be defined using the Interface Pattern.</td>
</tr>
<tr>
<td>CP5</td>
<td>Only Operations and State Variables defined for a Contract in a Contract Definition Viewpoint may be used in Contract Protocol Definition Viewpoints.</td>
</tr>
<tr>
<td>CP6</td>
<td>Each Contract Protocol, defined in a Contract Protocol Definition Viewpoint, must be a refinement of the collection of provided Interface protocols (defined in Protocol Definition Views) identified in a Contract Connections Viewpoint.</td>
</tr>
</tbody>
</table>

Note that the seven rules shown in Figure 178 are the minimum that is needed. Others could be added if required.

### 4.4.6.5. Contractual SoS Definition Viewpoint (CSDV)

The aim of the Contractual SoS Definition Viewpoint is defined in the Viewpoint Context View in Figure 179. For this view the main aim is to identify the contracts and the composition of the contractual SoS.
**Description**

The Viewpoint Definition View (VDV) in Figure 180 shows the Ontology Elements that appear on a Contractual SoS Definition Viewpoint.

The Contractual SoS Definition Viewpoint shows the Contractual SoS and the Contracts it is comprised of.

**Example**

An example view is shown in Figure 181, which conforms to the Contractual SoS Definition Viewpoint.
In the example ‘Contractual SoS Definition View’, we consider an ‘AV Contractual SoS’, comprising multiple ‘AV Device’ contracts and one ‘Transport Layer’ contract. The ‘AV Device’ contract comprises a ‘LE Device’, a ‘Browsing Device’ and a ‘Streaming Device’ contract.

The ‘LE Device’ contract states the functionality of an ‘AV Device’ necessary to guarantee a single leader – this is the focus in examples in this section. The other contracts that form the ‘AV Device’ contract ensure correctness of the media browsing and streaming functionalities of AV constituent systems (the ‘Browsing Device’ and ‘Streaming Device’ contracts respectively). The correct communication of data between contracts is ensured by the ‘Transport Layer’ contract.

This View along with Figure 184 fulfils rule CP1, with Figure 190 and Figure 191 (and views for the remaining contracts) fulfils rule CP2, with Figure 194 and Figure 195 (and views for the remaining contracts) fulfils rule CP3, and is a part of the minimum set of views contributing to rule CP5.

4.4.6.6. Contract Conformance Viewpoint (CCV)

The aim of the Contract Conformance Viewpoint is defined in the Viewpoint Context View in Figure 182. For this view the main aims are to identify the composition of the SoS in terms of the constituent systems and to identify the conformance relations between the contracts and constituent systems.
Description
The Viewpoint Definition View (VDV) in Figure 183 shows the Ontology Elements that appear on a Contract Conformance Viewpoint.

The 'Contract Conformance Viewpoint' includes a 'SoS' and the 'Constituent Systems' the SoS is composed of. The 'Contractual SoS', identified in the 'Contractual SoS Definition View', is included along with the 'Contracts' comprising the 'Contractual SoS'. The viewpoint also includes 'conforms to' relationships that detail the relationships between 'SoSs' and 'Contractual SoS', and 'Constituent Systems' and 'Contracts'.

Example
An example view is shown in Figure 184, which conforms to the Contract Conformance Viewpoint.


This View along with Figure 181 fulfils rule CP1.

4.4.6.7. **Contract Connections Viewpoint (CConnV)**

The aim of the Contract Conformance Viewpoint is defined in the Viewpoint Context View in Figure 185. For this view the main aim is to identify the connections between contracts.
Figure 185 - Context of the Contract Connections Viewpoint

**Description**
The Viewpoint Definition View (VDV) in Figure 186 shows the Ontology Elements that appear on a Contract Connections Viewpoint.

Figure 186 - Viewpoint Definition View showing the elements that appear on the Contract Connections Viewpoint (CConnV)

The *Contracts Connection* depicts the connections between the ‘Contracts’
comprising the SoS using the same connection concepts as in the Interface Connectivity Viewpoint. Contracts are connected between the ‘Ports’ and ‘Interfaces’ owned by the ‘Constituent Systems’ conforming to the Contract.

Example

An example view is shown in Figure 187, which conforms to the Contract Conformance Viewpoint.

![Contract Connections View](image)

Figure 187 - An example of a Contract Connections View showing the connections between two AV Device and one Transport contract

The figure shows that AV Device contracts are all connected to the Transport Layer contract through the same two interfaces (rec and send). These interfaces, not given here, may be defined using the Interfaces Pattern. These interfaces comprise simple sending and receiving operations – and are data agnostic. We also show that the LE Device, Browsing Device and Streaming Device contracts connect to the transport layer through the rec and send interfaces. In this figure, we give only two AV Device contracts; there may be many more.

This View is a part of the minimum set of views contributing to rule CP5. Although not defined for this example, each of the rec and send interfaces should each be defined using the Interface Pattern, and thus this view contributes to fulfilling rule CP4 and CP7.

4.4.6.8. Contract Definition Viewpoint (CDV)

The aim of the Contract Definition Viewpoint is defined in the Viewpoint Context View in Figure 188. For this view the main aim is to define the contract functionality. This in turn includes defining state variables, state invariants and operations, which includes defining the operation signature and pre/post-conditions.
Description
The Viewpoint Definition View (VDV) in Figure 189 shows the Ontology Elements that appear on a Contract Definition Viewpoint (CDV).

Figure 189 - Viewpoint Definition View showing the elements that appear on the Contract Definition View (CDV)
The 'Contract Definition Viewpoint' includes a single ‘Contract’; the ‘Operations’ owned by the Contract, several ‘State Variables’ and ‘State Invariants’, which are predicates constraining the ‘State Variables’.

**Example**

An example view that conforms to the Contract Definition View is shown in Figure 190.

![Contract Definition View](image)

Figure 190 - An example Contract Definition View

The Contract Definition View, realized here as a SysML Block Definition Diagram shows a partial definition of the ‘LE Device’ contract.

This partial definition names the private attributes (or values) and operations for the contract, then identifies two invariants over the values and gives three of the operations in more detail. The remaining operations are omitting for legibility reasons. The two invariants, ‘inv1’ and ‘inv2’, constrain the LE Device’s ‘mem’ and ‘otherLeaders’ state variables respectively. The definition of these invariants can be given in natural text, but in this example we use structured expressions defined in CML, which is the formal modelling notation developed specifically for modelling SoSs.

For example, within the leadership election, a device may choose to take the role of a leader, a follower, or to be undecided. The ‘changeClaim’ operation allows the device to change the role that it is going to take. It requires the new role (or claim) as a parameter. The precondition constrains the legal choice of new claims depending upon the current state of the ‘LE Device’. For example, the first clause of the pre-condition states that if the claim prior to operation invocation is set to <off>, then the new claim must be <undecided>. If the precondition holds, then the postcondition states that the ‘changeClaim’ operation will result in the ‘LE Device’s’ claim to be the one given as the parameter. The pre- and postconditions are, as with the state invariants, defined using CML.
As the Contract Pattern is notation agnostic, the (partial) example is defined using CML in Figure 191. This is equivalent to the SysML version in Figure 190.

These views (and views for the remaining contracts) along with Figure 181 fulfils rule CP2, is a part of the minimum set of views contributing to rule CP5, and along with Figure 194 and Figure 195 fulfils rule CP6.

```
process LE_Device = i : nat @
begin
  state
    id : NODE_ID := i
    mem: map NODE_ID to CS := {cid |-> mk_CS(<off>, 0) |cid in set node_ids \ {id}}
    inv dom mem = node_ids \ {id} and dom mem <> {}  
    highest_strength : STRENGTH := 0
    highest_strength_id : NODE_ID := 0
    inv highest_strength_id in set (dom mem union {id})
    leaders : nat := -1
    inv leaders <= card dom mem
    myCS : CS := mk_CS(<off>, 0)
    myNeighbours : seq of NODE_ID :=
      [i | i in set dom mem @ i <> id]
    isleader : bool := false
  operations
    write: NODE_ID * DATA ===> ()
    write(n, dat) ==
      ( 
        if is_TL_MSG(dat)
        then mem(n) := mk_CS(<off>, 0)
        else mem(n) := dat
      )
    pre i in set dom mem
    post mem(i) = dat or mem(i).c = <off>

    changeClaim: CLAIM ===> ()
    changeClaim(newc) ==
      ( 
        dcl currStr : STRENGTH := myCS.s @
        myCS := mk_CS(newc, currStr)
      )
    pre myCS.c = <off> => newc = <undecided> and
    myCS.c = <undecided> => newc = <leader> or
    newc = <follower> and
    myCS.c = <leader> => newc = <undecided> and
    myCS.c = <follower> => newc = <undecided>

    incStrength() ===>()
    incStrength() ==
      ( 
        if myCS.s < ulp
        then myCS := mk_CS(myCS.c, myCS.s+1)
      )
    pre myCS.s < ulp
    post myCS.s = myCS~.s + 1
end ...
```

Figure 191 - Contract Definition View defined using CML
4.4.6.9. **Contract Protocol Definition Viewpoint (CPV)**

The aim of the Contract Protocol Definition Viewpoint is defined in the Viewpoint Context View in Figure 192. For this view the main aim is to define the contract protocol – in terms of the ordering of external communications and internal state changes.

**Description**

The Viewpoint Definition View (VDV) in Figure 193 shows the Ontology Elements that appear on a Contract Protocol Definition Viewpoint.

have multiple Protocols governing their behaviour. Protocols often make use of concepts such as events, signals and the operations owned by the Contract. Such concepts have been deliberately omitted from the Protocol Definition Viewpoint, as they are dependent on the representation adopted for the realisation of the Viewpoint.

**Example**

An example of the Contract Protocol Definition Viewpoint is defined as a SysML state machine diagram in Figure 194.

![Figure 194 - Example of a Contract Protocol Definition View](image)

The protocol for this contract dictates the permitted ordering of operation calls for constituent system conforming to the contract. Briefly, the LE Device begins in an Off state, and may transition only to the On state. When in this state, the LE Device concurrently behaves in the Election state (transitioning between acting as (leader), (follower) and (undecided)) and the Listener state, (in which it updates its view of the states of the other devices according to the messages it receives).

The protocol is also defined in Figure 195 as a partial CML model. The protocol definition is defined using CML actions and is broadly in line with the SysML state machine in Figure 194.

These views (and views for the remaining contracts) along with Figure 181 fulfils rule CP3, is a part of the minimum set of views contributing to rule CP5, and along with Figure 190 and Figure 191 fulfils rule CP6. Although not defined for this example, each of the rec and send interfaces should each be defined using the Interface Pattern, including Protocol Definition Views, and thus this view contributes to fulfilling rule CP7.
D22.6 – Final Report on SoS Architectural Models (Public)

4.4.6.10. Summary

The Contract Pattern provides a set of views for the definition of the contracts that must be confirmed to by the constituent systems of a SoS. The contracts dictate the minimum behaviour that the constituents must provide to its environment.

The pattern includes five viewpoints: the Contractual SoS Definition Viewpoint identifies the contracts of the SoS; the Contract Conformance Viewpoint provides traceability in terms of the conformance of constituent systems to the contracts; the Contract Connection Viewpoint describes how the contracts are connected in terms of the interfaces owned by the constituents; the Contract Definition Viewpoint defines the contract operations and state variables; and the Contract Protocol Definition Viewpoint defines the behaviour of a contract in terms of event and operation ordering.

We demonstrate the use of both SysML and CML in the pattern description.

Figure 195 - Example Contract Protocol Definition View defined using CML
5. Summary

This deliverable contains the results of a study into modelling patterns for System of Systems (SoS) architectures. This deliverable has presented a discussion of architectural principles for both systems and SoSs, a classification of different SoS types, a collection of architectural styles and patterns for SoSs and overview of the current SysML to CML translation strategy. In this concluding section, we summarise the study documented in this deliverable.

As discussed in Section 1.2, architectural design is seen as an essential part of systems engineering and two key enablers for the production of an architectural design are architectural frameworks and modelling patterns. These two key enablers have guided the content of this report, as reflected in Sections 2 and 4 respectively.

In Section 2, the deliverable has discussed architectural principles for SoS by considering a number of sources to investigate:

- fundamental definitions of architecture and architectural framework.
- architectural techniques as they apply to systems.
- architectural techniques as they apply to systems of systems.

These concepts and techniques were then abstracted into a coherent set of definitions and principles.

Following the discussion on architectural definition and principles, in Section 3.1 the deliverable discussed the main characteristics for SoSs as described by several sources and described a number of examples of systems of systems, discussing them in relation to SoS types and recognized architectural patterns. These examples, presented in Section 3.2, were taken from different application domains in order to give the reader an understanding of different types of SoS and also to challenge the use of different architecture styles and patterns in relation to these application examples. The examples were also related to the architectural patterns presented in Section 4.3.

A number of architectural patterns were described in Section 4. Two different types of patterns were considered: architectural patterns that describe specific system architectures (such as a service oriented architecture) and enabling patterns whose use enables a number of systems engineering applications (such
6. References


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cocks 2006</td>
<td>Cocks, D. 'How should we use the term “system of systems” and why should we care?’ In Proceedings of the 16th INCOSE International Symposium 2006. INCOSE.</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>[D34.1 2012]</td>
<td>COMPASS. ‘D34.1 – Test Automation Support’. COMPASS Project; 2012</td>
</tr>
<tr>
<td>[D41.1 2013]</td>
<td>COMPASS ‘D41.1 - Accident Response Engineering Analysis Report Using Current Methods and Tools</td>
</tr>
<tr>
<td>[D42.1 2013]</td>
<td>COMPASS ‘D42.1 - A/V/HA Ecosystem Prototype Using Current Methods and Tools</td>
</tr>
<tr>
<td>[D43.2 2013]</td>
<td>COMPASS ‘D43.2 - Challenge Problem Analysis Using Existing Methods &amp; Tools</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Details</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RTCA DO-331]</td>
<td><strong>RTCA Inc.</strong> Model-Based Development and Verification Supplement to DO-178C and DO-278A</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>


[WDSL01] **W3C**, **Web Services Description Language v1.1**, 2001, Obtained from http://www.w3.org/TR/wsdl


Appendix A - Automated Model-Based Testing (MBT) and the Test Pattern

In this section we illustrate how the testing pattern introduced above can be instantiated in tools supporting automated model-based testing. As a “proof of concept” we will refer to the RT-Tester tool developed by Verified Systems International in cooperation with the University of Bremen. The utilisation of RT-Tester and its further development in the context of the COMPASS project is described in [D34.1 2013].

A.1. Case Study: Turn Indication Function

As a case study we consider the turn indication function of a vehicle providing left/right indication and emergency flashing by means of exterior lights flashing with a given frequency. Left/right indication is switched on by means of the turn indicator lever with its positions 0 (neutral), 1 (left), and 2 (right). Emergency flashing is controlled by means of a switch with positions 0 (off) and 1 (on). The requirements for the turn indication function are as follows (Table 8).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-001</td>
<td>Flashing requires sufficient voltage</td>
</tr>
<tr>
<td></td>
<td>Indication lights are only active if the electrical voltage is &gt; 80% of the nominal voltage.</td>
</tr>
<tr>
<td>REQ-002</td>
<td>Flashing with 340ms/320ms on-off periods</td>
</tr>
<tr>
<td></td>
<td>If any lights are flashing, this is done synchronously with a 340ms ON - 320ms OFF period.</td>
</tr>
<tr>
<td>REQ-003</td>
<td>Switch on turn indication left</td>
</tr>
<tr>
<td></td>
<td>An input change from turn indication lever state TurnIndLvr = 0 or 2 to TurnIndLvr = 1 switches indication lights left (output FlashLeft) into flashing mode and switches indication lights right (output FlashRight) off.</td>
</tr>
<tr>
<td>REQ-004</td>
<td>Switch on turn indication right</td>
</tr>
<tr>
<td></td>
<td>An input change from turn indication lever state TurnIndLvr = 0 or 1 to TurnIndLvr = 2 switches indication lights right (output FlashRight) into flashing mode and switches indication lights left (output FlashLeft) off.</td>
</tr>
<tr>
<td>REQ-005</td>
<td>Emergency flashing on overrides left/right flashing</td>
</tr>
<tr>
<td></td>
<td>An input change from EmerFlash = 0 to EmerFlash = 1 switches indication lights left (output FlashLeft) and right (output FlashRight) into flashing mode, regardless of any previously activated turn indication.</td>
</tr>
<tr>
<td>REQ-006</td>
<td>Left-/right flashing overrides emergency flashing</td>
</tr>
<tr>
<td></td>
<td>Activation of the turn indication left or right overrides emergency flashing, if the latter has been activated before.</td>
</tr>
<tr>
<td>REQ-007 Resume emergency flashing</td>
<td>If turn indication left or right is switched off and emergency flashing is still active, emergency flashing is continued or resumed, respectively.</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>REQ-008 Resume turn indication flashing</td>
<td>If emergency flashing is turned off and turn indication left or right is still active, the turn indication is continued or resumed, respectively.</td>
</tr>
<tr>
<td>REQ-009 Tip flashing</td>
<td>If turn indication left or right is switched off before three flashing periods have elapsed, the turn indication will continue until three on-off periods have been performed.</td>
</tr>
</tbody>
</table>

Table 8- Requirements list for the turn indication function

A.2. RT-Tester Test Modelling Approach

For automating model-based test campaigns RT-Tester encourages users to provide a small number of SysML models from where the views of the Test Pattern can be derived in an automated way, finally leading to test sets that can be automatically executed by test procedures, each exercising one or more test cases of the set. The top-level element of this test model is represented by a block definition diagram representing both test environment (TE) and system under test (SUT), as shown in Figure 196.

A.3. Testing Context View

The System block is decomposed into TE and SUT blocks, as shown in Figure 197.
From this internal block diagram the first part of the ‘Testing Context View’ can be derived:

- The interfaces shown as item flows and signal events (not used in this case study) define the level of abstraction to be used for the test set to be created from this model: these interfaces are the observable SUT outputs and writable inputs that may be accessed by the TE.
- RT-Tester allows for SysML properties and signal events to be exchanged between SUT and TE model components. RT-Tester provides interface modules mapping their valuations onto concrete software or hardware interfaces and vice versa.

In a software integration test for the turn indication system the turn indication lever values and the status of the emergency switch may be passed to the SUT, for example, by means of shared variables. The SUT outputs (left-hand side lamps on/off, right-hand side lamps on/off) can also be represented by Boolean output variables of the SUT. In a HW/SW integration test interface modules would map the turn indication lever status and the emergency flash button to discrete inputs to the SUT. In a system integration test the actual voltage placed by the SUT on the indication lamps would be measured.

The interface abstraction required for the test level is specified by a signal map that associates abstract SysML model interfaces with concrete interfaces of the test equipment.

The visualisation of the ‘Testing Boundary’, the ‘Testing Need’ and other aspects identified in Section 4.4.2.5 are optional representations that may be helpful for manual inspection of the ‘Testing Context View’ but are not needed for the automated test generation process. If they are needed, they have to be created in a manual way. The relevant information for automated MBT lies in the interface between the TE and SUT block.
The structural view on the SUT has to be decomposed further, until each block is associated with a sequential behaviour. For the case study discussed here, the SUT is further decomposed into two concurrent functions as depicted in Figure 198.

Functional component FLASH_CTRL performs the decisions about left/right indication or emergency flashing. Component OUTPUT_CTRL controls the flashing cycles and switches of indication lamps if the voltage gets too low.

### A.4. Test Behaviour View

In the terms of the Test Pattern introduced above test automation process is focused around the 'Test Behaviour View', and several aspects of other views will be populated from there. A 'Test Behaviour View' in RT-Tester consists of the behavioural specification portions of the SysML test model: instead of modelling just the portion of SUT behaviour to be exercised in one specific test case, the RT-Tester approach encourages users to specify complete aspects of SUT behaviour, and rely on the test generator to extract the relevant test cases and associated data from such a model.

For our case study the two functional components introduced above have the behaviour specified by the state machines shown in Figure 199 and Figure 200.
The FLASH_CONTROL component makes the decision whether no flashing, left/right flashing or emergency flashing should be performed and passes this information on to component OUTPUT_CTRL via interface Left/Right.

- As long as the emergency flash switch has not been activated, Left/Right are set according to the turn indication lever status. This is specified in do activity doEmerOff.
As soon as the emergency flash switch EmerFlash is switched on, Left/Right are set as specified in sub-state machine EMER_ON.

Left/Right are both set to true and the state machine remains in control state EMER_ACTIVE.

When the turn indication lever is changed to left or right position, emergency flashing is overridden, and left/right indication is performed.

Function OUTPUT_CTRL sets the SUT output interfaces FlashLeft and FlashRight. The indication lamps are switched according to the internal interface state Left/Right, if the voltage is greater than 80% of the nominal voltage. If the lamps have been on for 340ms they are switched off and stay off until 320ms have passed. A counter FlashCtr is maintained: if the turn indication lever is switched from left or right back to the neutral position before 3 flashing periods have been

Figure 200 - Hierarchic state machine OUTPUT_CONTROL
performed, left/right indication will remain active until the end of these 3 periods.

A.5. Test Case Generation

With the behavioural model at hand in the ‘Test Behaviour View’, test cases can be identified in an automated way by applying different model coverage strategies. The full set of these strategies is described in [D34.1 2013]; here we give an example of the transition coverage test strategy, which requires that every state machine transition should be covered at least once in a test suite. For sub-machine FLASHING (Figure 200) the logical condition for covering the ON → OFF transition can be expressed in Linear Time Logic LTL by

\[
\text{FINALLY (ON and} \\
((\text{Left or Right) and Voltage > 80}) \text{UNTIL} \\
((\text{Left or Right) and Voltage > 80 and timer > 340}))
\]

Informally expressed, this means “finally state ON is reached, and Left or Right remain active and the Voltage stays above 80% of the nominal voltage until more than 340ms have passed since entering state ON” (‘timer’ is an internal timer that is automatically reset when state ON is entered, so that the relative time since entering this state can be measured).

The formula above is called a symbolic test case because it specifies the test objective without giving the concrete test data. The constraint solver which is part of RT-Tester can determine concrete timed input sequences to the SUT from these formulas, by calculating traces that are consistent with the model and at the same time fulfil the symbolic test case formula (see [D34.1 2013]). For this example an input trace like

1. \((t = 0, \text{Voltage = 81, TurnIndLvr = 1})\)
2. \((t = 345, \text{Voltage = 85, TurnIndLvr = 0})\)

represents a solution for this test case. At the end of the symbolic test case generation process a ‘Test Set’ of symbolic test cases has been identified that is “useful” for covering the model and “sufficient” for the model coverage typically required by standards like [RTCA DO-331]. Since each test case has been derived from the model, the ‘TestableElement’ associated with each test case are automatically identified as well. RT-Tester generates unique test case identifiers, and a ‘TestDescription’ is produced from the symbolic test case formulas in natural language. Optionally the natural language description can be extended by the users.

A.6. Automated Traceability Data Generation

In SysML model elements can be associated with requirements by means of the <<satisfy>> relationship. The interpretation is that every computation covering the
model element under consideration contributes to the realisation of the requirement. Consider, for example, ‘REQ-002 Flashing with 340ms/320ms on-off periods’. This requirement is tested whenever a computation covers the ON \(\xrightarrow{}\) OFF and OFF \(\xrightarrow{}\) ON transitions in sub-machine FLASHING. Therefore these transitions are linked to REQ-002, and the test case specified above contributes to the verification of REQ-002. The Traceability matrix relating requirements and model elements is shown in Table 9.

<table>
<thead>
<tr>
<th>Name</th>
<th>Satisfied By</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-001 Flashing requires sufficient voltage</td>
<td>«Constraint» OUTPUT_CTRL.Constraint_002 (Model::Constraints) FINALLY (Voltage &lt;= 80 and (Left</td>
</tr>
<tr>
<td>REQ-002 Flashing with 340ms/320ms on-off periods</td>
<td>«Transition» [ON - OFF] «Transition» [OFF - ON]</td>
</tr>
<tr>
<td>REQ-003 Switch on turn indication left</td>
<td>«Constraint» OUTPUT_CTRL.Constraint_003 (Model::Constraints) FINALLY (FlashLeft == 1 and FlashRight == 0)</td>
</tr>
<tr>
<td>REQ-004 Switch on turn indication right</td>
<td>«Constraint» OUTPUT_CTRL.Constraint_004 (Model::Constraints) FINALLY (FlashLeft == 0 and FlashRight == 1)</td>
</tr>
<tr>
<td>REQ-005 Emergency flashing on overrides left/right flashing</td>
<td>«Atomic State» EMER_ACTIVE (Model::SYSTEM::SystemUnderTest::FLASH_CTRL.EMER_ON) «Constraint» FLASH_CTRL.Constraint_002 (Model::Constraints) FINALLY (EMER_ACTIVE and TurnIndLvr &gt; 0)</td>
</tr>
<tr>
<td>REQ-006 Left-/right flashing overrides emergency flashing</td>
<td>«Atomic State» TURN_IND_OVERRIDE (Model::SYSTEM::SystemUnderTest::FLASH_CTRL.EMER_ON)</td>
</tr>
<tr>
<td>REQ-007 Resume emergency flashing</td>
<td>«Transition» [TURN_IND_OVERRIDE - EMER_ACTIVE]</td>
</tr>
<tr>
<td>REQ-008 Resume turn indication flashing</td>
<td>«Constraint» FLASH_CTRL.Constraint_001 (Model::Constraints) FINALLY (EMER_ACTIVE and not EmerFlash and TurnIndLvr &gt; 0)</td>
</tr>
<tr>
<td>REQ-009 Tip flashing</td>
<td>«Constraint» OUTPUT_CTRL.Constraint_001 (Model::Constraints) Voltage &gt; 80 and not (Left or Right) and ((Left1 and not Right1) or (not Left1 or .Right1)) and FlashCtr &lt; 3)</td>
</tr>
</tbody>
</table>

Table 9 - Traceability matrix relating requirements to model elements

In some situations the computations realising a requirement cannot be appropriately defined by just indicating model elements. Instead, LTL formulas
are used in SysML constraints to specify these computations is a comprehensive way. Consider, for example, ‘REQ-008 Resume turn indication flashing’. The constraint shown in the traceability matrix above expresses that each computation contributing to REQ-008 shall finally reach state EMER_ACTIVE (that is, emergency flashing is on), and the emergency flashing switch is then already turned off (so that we transit into state EMER_OFF), but the turn indication lever requires left or right flashing. Again, the constraint solver can calculate solutions for these constraints, resulting in concrete test cases for the associated requirements. Figure 201 shows a traceability matrix as provided by RT-Tester for relating requirements to test cases («verify» or «validate» relationship).

A.7. Test Procedures, Test Records and Test Schedules

In the previous sections it was shown how a ‘TestSet’ of ‘TestCase’ can be automatically derived from ‘TestBehaviour’, and each test case traces to one or
more requirements by means of the «verify» or «validate» relation\(^7\). When generating test data for a set of symbolic test cases, users specify which concrete cases should run in one test procedure. Test procedures are the executable units exercising one or more test cases against the SUT, thereby producing instances of 'TestRecord'. Several test procedures can be arranged in a 'TestSchedule' (also called a test suite). Test schedules can be run automatically against the SUT.

\(^7\) The distinction between «verify» and «validate» has to be made manually, since many test cases are suitable for both verification and validation.
Appendix B - Refinement in SysML

In this appendix, we describe a refinement calculus for SysML that aims to support the formal analysis of SysML models. The main requirement for the investigation of analysis of SysML model is the formal semantics that supports the specification and verification of properties. The integrated semantics of SysML diagrams is presented in the deliverable D22.4. The results described in this deliverable and in D22.4 allow the investigation of formal properties relevant to the patterns through a widely used technique (formal refinement) suitable for the verification of models.

In Section B.1, we describe the current state of the formal semantics of SysML. Section B.2 briefly discusses the guideline of usage required by the semantics, Section B.3 lifts the notions of refinement inherited from CML to SysML, Section B.4 presents SysML extensions necessary to support a SysML refinement calculus, and Section B.5 proposes an initial catalogue of SysML refinement laws.

B.1. Formal semantics of SysML models

A subset of SysML diagrams and constructs as well as specific patterns of usage have been given a formal semantics based on the CML notation. The formal semantics is given in terms of translation functions from SysML constructs to CML elements, and a top-down translation strategy that guides the application of the translation rules to SysML models.

The subset of SysML covered by the translation rules are block definition diagrams, internal block diagrams, state machine diagrams, activity diagrams and sequence diagrams. For block definition and internal block diagrams, the following constructs are supported: Blocks, operations and signals, ports, provided and required interfaces, associations, composition, and generalisation.

For state machine diagrams, the supported constructs are: simple and composite states, regions, final states, initial and junction pseudostates, fork and merge pseudostates, transitions, and completion and deferred events. For activity diagrams, the translation strategy supports: control nodes (Decision, ActivityFinal, FlowFinal, Fork, Join, Initial, Merge), simple actions, accept event and send signal actions, control flow, interruptible activity region. Finally, for sequence diagrams the supported constructs are: lifelines, synchronous messages, asynchronous messages, reply messages, interaction use, gates (for interaction use only), execution specification, state invariants and alternative, option, loop, break, strict and parallel operators for combined fragments.

A key feature of this translation strategy is that it is compositional, that is, the translation of a SysML constructs depends solely on the elements of the construct. The semantics of each of the three behavioural diagrams, namely state machine, activity and sequence diagrams is given in terms of independent
translation rules and strategies. The overall behaviour of a SysML model is given by a set of translation rules and a translation strategy based on a guideline of usage; these rules and strategy rely on the previously defined semantics of the behavioural diagrams.

The application of the translation strategies to SysML diagrams (conforming to the subset of constructs mentioned) produces a CML model composed by a number of definitions. The root of the CML model is a process appropriately named which defines the overall behaviour of the diagram.

The model of a block consists of two main process definitions:

- Event treatment process: this process receives events from an external environment, adds them to a pool and selects the events (one at a time) to be processed by the state machine;
- Bare process: this process exposes the interfaces (operations and signal) that a block provides and requires; it relies on other processes that model different aspects of the block, such as its ports and behaviours.

The model of a state machine diagram consists of a process that models the processing of a particular event by the state machine.

Regarding the translation of sequence diagrams, the general use only requires one main process that is composed of CML actions determined by the lifelines and their interaction fragments. These lifelines are gathered in a main CML action that represents the whole diagram. The special case is when we use the Interaction Use resource (a sequence diagram inside another through reference). In this situation we have at least three processes: one for the reference, another for the enclosing diagram and the main that compose the other two. If more than one reference is used, more processes will be needed.

Similarly, the translation of activity diagrams produces a number of CML definitions including a root CML process that specifies the overall behaviour of the diagram. Basically, all elements in the activity diagrams are indexed, and the nodes are translated to CML actions of the main process. These nodes are arranged according to indexes of the edges that connect them.

The integrated semantics of SysML models is defined in terms of specific translation rules that generate bare models of the blocks (based on block definition and internal block diagrams) and the translation strategies for state machine, activity and sequence diagrams. The information contained in the block definition and internal block diagrams provide the context in which the behaviours specified by the remaining diagrams can be composed. This is achieved by composing the bare models of the blocks, and the root processes of each behavioural diagram in the SysML model to produce the overall behaviour of the SysML model. The composition of these processes reflects the structure of the SysML model.
B.2. Guidelines of usage

Because the semantics of SysML models is given in terms of CML specifications, it is possible to apply all the analysis techniques available to CML to the SysML models. Some of the analyses that can be carried out are described below.

- Verify that the system specified by a SysML model does not reach a state where no further progress is possible;
- Verify that a scenario specified by a sequence diagram is possible (or not) in the SysML model;
- Verify that an updated model maintains the behaviour of the source model;
- Verify that the system does not contain an infinite loop;
- Verify that a system is a correct implementation of a specification.

However, due to the SysML semi-formal semantics (inherited from UML), where a wide range of possibilities of meaning may take place, it is not feasible in the context of this project to define a unique semantics that cover all the semantic variation points of the language. Thus, we need to define a set of guidelines of usage wherewith we can extract unique meanings from the models and where we can generate the correspondent CML models.

A simplified example of a guideline of usage of these diagrams is described below:

- Each block may have an associated state machine diagram (of the same name) specifying the overall behaviour of the block;
- A block definition diagram must exist containing a root block representing the whole system and it must relate to the other leaf blocks that represent the constituent systems through a composition relationship;
- The possible communications among the constituent blocks must be specified only in the internal block diagram of the root block;
- Blocks in block definition diagrams may be connected only by generalisation, composition and aggregation. Association can only be used in internal block diagrams;
- Each composition/aggregation root must either have an internal block diagram or we assume that everyone talks to everyone;
- Multiplicity must be a finite fixed number;
- Each operation is defined either by an activity diagram, or by a CML Action Language specification or, implicitly, by the block's state machine diagram;
- Sequence diagrams specify particular scenarios the model must or must not allow;

A complete set of guidelines of usage required by the formal semantics can be found in the deliverable [D22.4 2013]. In the next sections, we discuss our work on extending an important CML analysis technique, namely refinement, to SysML. Firstly, we lift the CML notion of refinement to establish suitable refinement relations for the different elements of SysML (e.g., blocks, states).
Next, we propose extensions to SysML that support the effective use of refinement, and finally, we propose a number of refinement laws motivated by a simple case study.

B.3. Refinement in SysML

Informally, our notion of refinement for SysML models compares the two blocks that define the systems with respect to their operations and signals. Essentially, if a block $A$ is refined by a block $B$, the following properties must hold:

- $A$ and $B$ must accept exactly the same public signals;
- $A$ and $B$ must accept exactly the same public operations;
- $A$ and $B$ must have exactly the same public properties;
- for each public operation of $A$, if its return value is nondeterministically chosen from a set $S$, the same operation on block $B$ must return a value that is nondeterministically chosen from a subset of $S$;
- for each property of $A$, if its value is nondeterministically chosen from a set $S$, the same property on the block $B$ must have a value nondeterministically chosen from a subset of $S$.

This refinement relation is induced by the CML semantics of SysML and corresponds to the refinement relation of CML process.

First of all, since process refinement is compositional in CML, and the main SysML elements (blocks, state machines and activities) map to processes used to define the CML process that defines the system model, we can refine the models of the individual diagrams to refine the SysML model as whole.

Next, we formalise the notion of refinement for blocks and state machines.

Definition B.1. Block refinement
Let $M$ be a SysML model, and let $B_1$ and $B_2$ be blocks of $M$, then $B_1 \sqsubseteq_{\text{Block}} M \sqsubseteq_{M,B} B_2$ if, and only if, $t_{\text{model}}(M).B_1 \sqsubseteq_{P} t_{\text{model}}(M).B_2$.

That is, block $B_1$ is refined by block $B_2$ (written $B_1 \sqsubseteq_{M,B} B_2$, where $M$ is the model that contains $B_1$ and $B_2$, and Block indicates that the refinement applies to blocks) if, and only if, the CML process $B_1$ that models the block $B_1$ is refined by the process $B_2$ that models $B_2$. With the view that a system is specified by a block in a SysML model, block refinement as formalised in Definition B.1 is the main relation that must be verified to establish refinement between systems.

Data-refinement in SysML is defined similarly to behavioural refinement by lifting CML data-refinement, which is based on forward simulation.

Definition B.2. Forward Simulation
A forward simulation ($\preceq$) between blocks $B_1$ and $B_2$ of a SysML model $M$ is a relation $R$ between $B_1.\text{PrivateProps}$ and $B_2.\text{PrivateProps}$ if, and only if, $R$ is a forward simulation between the processes $t_{\text{model}}(M).B_1$ and $t_{\text{model}}(M).B_2$. 

301
Unlike state components of CML processes, properties of blocks are not necessarily encapsulated (private). For this reason, forward simulation in SysML is defined with respect to private properties of the blocks.

**Definition B.3. State machine refinement**

Let \( M \) be a SysML model, and let \( S_1 \) and \( S_2 \) be state machines of \( M \), then \( S_1 \sqsubseteq_{M, \text{Stm}} S_2 \) if, and only if, \( t_{\text{model}}(M).S_1 \sqsubseteq_P t_{\text{model}}(M).S_2 \).

That is, a state machine \( S_1 \) is refined by another \( S_2 \) (written \( S_1 \sqsubseteq_{M, \text{Stm}} S_2 \), where \( M \) is the model that contains \( S_1 \) and \( S_2 \), and \( \text{Stm} \) indicates that the refinement applies to state machines) if, and only if, the CML process \( S_1 \) that models the state machine \( S_1 \) is refined by the process \( S_2 \) that models \( S_2 \).

Notions of refinement for states (\( \sqsubseteq_{M, \text{State}} \)), regions, transitions and actions are similarly defined. For states, the observations that are preserved by refinement are the activation and deactivation of the top state (that is, the substates are not observable), and the signals and operation calls performed inside the state. The observations of regions are the activation and deactivation of the region and the signals and operation calls performed inside the region. Transition refinement preserves the observation of activation and deactivation of states as well as the signals and operation calls performed by the transition.

As indicated in the previous section and further explained in Section B.4, a subset of CML is used as action language for SysML. This subset excluding signals, operation calls and return statements retains the original CML semantics (except that they are enclosed in a variable block that models a local copy of the shared state). For this reason, CML refinement laws for such statements can be reused in the refinement of SysML models.

**B.4. SysML extensions**

In general, we wish to prove that an abstract model, where possibly no particular design has been chosen, is refined by a more concrete model in which some design decisions have been taken. In SysML, the more concrete model often adds new operations to the abstract model in order to implement particular designs. This, however, makes the refinement invalid as new operations are now observable in the concrete model. The extra operations should in fact be internal, and used solely to implement behaviour specified in the abstract model.

We motivate and illustrate our extensions and refinement laws through a simple example of a chronometer that records seconds and minutes, and accepts a **tick** signal that increments the chronometer and a **time** operation that queries the recorded time. The example consists of two distinct models, one abstract, depicted in Figure 202 and one concrete, shown in Figure 204, related by refinement. Whilst the abstract model is centralised, the concrete one has two components, one recording the seconds and the other recording the minutes. The components of the concrete model cooperate to realise the behaviour specified in the abstract model.
Figure 202. Block definition diagram of abstract model.

Figure 202 shows the block definition diagram of the abstract model; it declares a single block `AChronometer` with two private properties `sec` and `min`, both of type `Integer`, and a port `port` that provides the operations and signals in the interface `ChronometerI`. This interface is also defined by a block and contains an operation `time` that returns a value of type `Time`, and a signal `tick` that models the passing of time. The type `Time` is a datatype with two components, `min` and `sec`, that encode a time instant in minutes and seconds.

Figure 203. State machine diagram of the abstract model.

Since the block `AChronometer` is simple, there is no internal block diagram specifying its internal structure. The remaining diagram in the abstract model is the state-machine diagram shown in Figure 203; it contains a single simple state. When the state machine is started, the properties `min` and `sec` are initialised to 0, and the state `State` is entered. When the state is active, either the internal transition triggered by `time` is executed, or the transition triggered by `tick` is executed. The first models the treatment of a call to the operation `time` and returns a value of type `Time` built from `min` and `sec`, whilst the second models the passing of time and increments the block’s properties.

Figure 204. Block definition diagram of concrete model.
The concrete model is formed by four diagrams: one block definition diagram, one internal block diagram and two state-machine diagrams. The first is shown in Figure 204; it declares three blocks **CChronometer**, **Min** and **Sec**. The first is composed of the other two as indicated by the composition relation (arrow with a black diamond). The block **CChronometer** is similar to the block of the abstract model except that it has no properties. These are distributed in the components **Min** and **Sec**. The block **Min** has a single private property and two ports, **ip** and **ip_aux**, that both provide the operations in the interface **MinutesI**. The provided and required interfaces of a port are the sets of operation calls and signals that the block, respectively, receives and sends through the port.

The block **Sec** also has a single property, but two ports, **port** and **ip_conj**. The first is identical to the port of the block **CChronometer**, whilst the second is complementary to the port **ip** of **Min** and requires the operations in the interface **MinutesI**. The block definition diagram has some extra annotations (hiding and through port), which support refinement and are explained in this section.

The internal block diagram of the concrete model (Figure 205) shows the composite block **CChronometer** and its components (marked with <<part>>); it specifies that the port **port** of **Sec** is connected to the port of **CChronometer**, and that the ports **ip** and **ip_conj** are connected to each other. Finally, the blocks **Sec** and **Min** have each one state machine (Figure 206 and Figure 207). The state machine of **Sec** is called **SecMain** and is similar to the state machine of the abstract model, except that it delegates the operations involving minutes to the block **Min**. These operations are treated by the state machine **MinMain** that contains a single state with two internal transitions that react to a call to the operations **minsReq** and **inc**. The first returns the value stored in **min**, and the second increments it.

![Figure 205. Internal block diagram of concrete model.](image-url)
In our example, we wish to show that the block \texttt{AClockometer} is refined by \texttt{CChronometer}. However, based solely on the pure SysML model, it is not possible to verify this refinement since block \texttt{CChronometer} clearly offers more operations than \texttt{AClockometer}: \texttt{inc}, \texttt{minsReq} and \texttt{reset} from block \texttt{Min} in Figure 204. These operations are used to implement the operation \texttt{time}, and are not meant to be visible outside the block \texttt{CChronometer}, that is, they are meant to be internal.

Moreover, since some of the ports of internal parts can be left unconnected, the operations and signals they offer are not called by another part, and simply making them internal, could lead them to occur spontaneously. For this reason, there needs to be a way of making them unavailable when hidden.

Finally, SysML does not provide adequate support for specifying abstract behaviours: both state machines and activities define very concrete models, and the fact that their action language is undefined is also a hindrance. The use of a programming language to define the action does not address this issue; as it does not provide support for abstract specifications.

We address the problems above through five extensions to SysML: hiding, restrictions, alphabets, plugs and the definition of an action language. The first extension supports the specification of internal signals and operations, the next...
three can be used to make certain signals and operations unavailable, and the fifth adds support for abstract specifications.

In order to specify that certain operations and signals are internal to a block, we propose the use of the hiding extension. A set of operations and signals represented by a SysML interface is hidden in a block by creating a dependency between the block and the interface, and adding a **hiding** comment to the dependency as shown in Figure 204. The semantics of this extension is given by the hiding operator of CML, which makes a set of channels internal to a process, and therefore, independent from external influences.

As already mentioned, any internal operation or signal that is offered but not used can occur spontaneously, which, in turn, leads to an infinite loop of internal behaviours. Therefore, only operations and signals that are used as specified by the internal block diagram can be made internal.

This restriction, however, is too strong as unused operations are often assumed to be unavailable. In fact, in our example, none of the extra operations of **CChronometer** can be hidden due to this restriction: some of them (reset) are not used at all, and the others are used only through a particular port of Min. The next three extensions provide mechanisms to indicate that an operation or signal is unavailable under certain situations (and can, therefore, be hidden).

Operations and signals of a block can be called by referring directly to an instance of the block, or through the ports that provide them. In order to support the specification of operations and signals that are only used through ports or (directly) through the block, we propose the use of restrictions.

Restrictions are represented by a **through ports** or **through block** comment linked to an operation, signal or interface to indicate that it is offered only through ports or only through the block. If there is no comment, it remains available through both. Figure 204 illustrates the use of restrictions in the realisation between the block Min and the interface MinutesI. The semantics of a restriction that declares an operation O only available through ports is given by a reduction of the alphabet of the process that models the block. This reduction removes all communications that allow calls to the operation directly to the block.

Alphabets specify which operations and signals of a block are available when it is used as a part of another block. This extension is represented by a SysML comment that lists the used operations and signals and is associated with particular instances of blocks (parts). Alphabets must be connected to part (and not blocks) because they specify restrictions over the use of a block as a part. A block may be used in different contexts with different alphabets. In our example, to prevent the part min shown in Figure 205 from offering the operation reset, it is annotated with an alphabet containing minsReq and inc.

At this point, the operation reset can be hidden because it is not offered by the block **CChronometer** or its parts, but the remaining operations of Min cannot.
They are offered on ports `ip` and `ip_aux`, but only used on `ip` as indicated by the connector between `ip` and `ip_conj` in Figure 205. Before these operations are made internal, the unused port `ip_aux` must be disabled. This is achieved by means of the plug extension, which allows us to mark a port as unused (plugged). A plug is specified by a comment linked to the port that is unavailable. Similarly to alphabets, this annotation must be placed on a port of a part since it does not affect a block in general, but only a particular use of a block. In our example, the port `ip_aux` is plugged as shown in Figure 205.

Finally, the problem of supporting abstract specifications is addressed by the use of a subset of CML as action language in state machines (and activities). This subset includes the CML statements (like the specification statement for instance), as well as sequential composition, external and internal choice, interleaving and guarded statements. These are the basic CML action constructors, except for those that involve communication (prefixing and parallel composition) since the communication paradigm of SysML (asynchronous) is different from that of CML (synchronous) and is already supported by signals and operation calls. In our example, the only statements that are used are assignments, specification statement and sequential compositions as shown in Figure 203, Error! Reference source not found. and Figure 207. The complete syntax of CML statements is in D23.1.

**B.5. Refinement laws in SysML**

In this section, we describe how the Circus refinement strategy can be applied to SysML models and present a few laws that support the strategy. These laws fall into two main groups: refinement laws that rely solely on existing SysML constructs, and laws that use alphabets, restrictions, plugs or hiding.

Figure 208 presents the iterative refinement strategy of Circus. In each iteration, initially, a centralised abstract process is data refined to introduce concrete data models, next the actions of the process are refined to introduce parallelism, and finally the process is partitioned into one or more processes that interact with each other to implement the abstract process. Each of the new processes may become the object of a subsequent iteration of the strategy.

We illustrate the use of this refinement strategy for SysML and the new laws through a simple example that verifies that the distributed concrete model shown in Figure 204 is a refinement of the abstract model in Figure 202.
The first phase of the refinement strategy is supported in SysML by simulation laws that distribute a forward simulation (see Definition B.2) through a SysML model. Since the data model of our concrete specification is the same as that of the abstract specification, this phase is not required in this simple example.

In the second phase, we start by introducing local auxiliary behaviours in the abstract model that are initially not used via state machine refinement laws. Namely, the local hidden operations \texttt{minsReq} and \texttt{inc} are introduced by the Law B.1 presented below.

\textbf{Law B.1. Local operation introduction}

\begin{align*}
\text{\texttt{minsReq} and \texttt{inc} are introduced by the Law B.1 presented below.}
\end{align*}

This law takes a block and an operation, introduces the operation in the block as a private operation, and hides it. The resulting block is identical to the original because the new operation is only available internally and is not used. Whilst
this seems useless, further laws can take advantage of the availability of the local operation to replace behaviours by calls to it.

Still as part of the second phase of the strategy, we introduce some of the structure of the design. In our example, the single state in the abstract state machine is refined into a composite state with two regions using the Law B.2 shown below (the first region corresponds to SecMain in Figure 206, and the second contains a state that offers the behaviours associated with the local operations. This does not modify the behaviour of the state machine because the newly introduced behaviours are triggered by unused local operations.

**Law B.2. Region introduction**

This law takes a composite state with a single region containing any number of substates and transitions, and refines it into a composite state with two regions: the first is the original region, and the second is an empty region. This is possible because the two regions are executed in parallel, and the empty region does not introduce new behaviours observable outside the composite state.

In the third phase, the block AChronometer is partitioned in two using Law B.3. This law is a block refinement law that takes a simple block with two ports, p1 and p2, and a state machine that at the top level has two regions, R1 and R2. It refines the simple block into a composite block with the same two ports, but whose parts are two new blocks, Block1 and Block2, each with two ports (e.g., p1 and ip1), and each with its own state machine derived from one of the regions R1 and R2. In this law, I1 and I2 represent both the provided and required interfaces of the port.

**Law B.3. Block decomposition**
The provisos of this law guarantee that no new operations or signals are introduced and that their treatments (in the state machine) are independent and, therefore, can be separated. That is, this law can be applied as long as the a subset of the operations and signals (the external ones) of the original block are partitioned in the interfaces $I_1$ and $I_2$ (proviso 1), the transitions of the two top regions of the state machine have no triggers in common and the two regions do not share block properties (proviso 2), the provided items (operations and signals) of $I_1$ are not used in the triggers of the transitions of region $R_2$ and the required items of $I_1$ are not used in the actions of the states and transitions in $R_2$. 

\[
\begin{align*}
\text{where} & \\
\text{Block1} & \cap \text{Block2} = \emptyset \land \text{Block1} \cup \text{Block2} = \text{Block} \\
\text{provided}(I_1) & = \text{Block1} \cap \text{used}(R_2) \land \text{required}(I_1) = \text{Block2} \cap \text{used}(R_1) \\
\text{provided}(I_2) & = \text{Block2} \cap \text{used}(R_1) \land \text{required}(I_2) = \text{Block1} \cap \text{used}(R_2)
\end{align*}
\]
(proviso 3), and the provided items of I2 are not used in the triggers of the transitions of region R1 and the required items of I2 are not used in the actions of the states and transitions in R1 (proviso 4).

Each block has an event pool where received events (operation calls and signals) are stored for processing. The proviso 2 of Law B.3 guarantees that it is possible to partition the event pool of the block into two parts: one containing only events that may be consumed by the first region, and the other containing the events that may be consumed by the second region. Since the order in which events are sent to the state machine is non-deterministic, it is not possible to distinguish the two pairs of event pools and state machines from the original pair, thus allowing the block to be decomposed in two.

The two new blocks produced by the Law B.3 partition the operations and signals of the original block, and each has two ports; for instance, in Block1 they are p1 and ip1. The ports p1 and p2 are identical to those of the original block and are linked by a connector to the corresponding ports of the composite block. The connected ports ip1 and ip2 are introduced to allow one part to call operations of the other, which accounts for the use of block operations in the original state machine. The interfaces II1 and II2 of these internal ports are such that they contain as provided items those operations and signals of the associated block (Block1 or Block2) that are used by the region associated with the other block, and contain as required items those that are used by its associated region. Both of these interfaces are hidden.

In a second iteration of the refinement strategy, standard CML refinement laws are used to (1) introduce a local variable aux initialised with min in the behaviour of the transition triggered by time (see Figure 203), and (2) replace min in the record constructor mk_Time by the local variable. Finally, a law that allows an action to be replaced by an operation call is applied twice, once to replace the assignment aux := min by a call to minReq via port ip_conj, and again to replace the assignment min := (min + 1) mod 60 by a call to inc through the same port.

The soundness of these laws can be verified using the CML models induced by our semantics, and our notions of refinement.