OSRA Specification and Rationale

Peter Mendham and the COrDeT-3 Consortium

ADCSS - 27th October 2014

Overview

- Aim and Role of the specification
- Document overview
- Architecture Overview
- Component Layer specification
  - Example: M&C in the OSRA
- Interaction Layer Specification
- Execution Platform Specification

bright ascension

OSRA Specification

- A normative specification of the Onboard Software Reference Architecture
- Is also required to
  - Introduce and explain architectural principles
  - Rationalise architectural design choices
  - Acts as an answer to original SAVOIR-FAIRE User Needs
- Too much for one document
  - Focus would be lost
- Split into two:
  - **Specification** and **Rationale**
- The specification relates closely to other documents
  - Rationale
  - Meta-model specification (and meta-model)
  - Training material

Specification – Core Documents

- OSRA Specification (D02-SPEC)
  - Introduction
  - Key concepts
  - Informative guide to SCM
  - Normative specification of interfaces
- SCM Specification (D03)
  - Normative specification of meta-model
  - Specification of model exchange format
- Meta-Model (M01)
  - Normative meta-model

Supporting Material

- OSRA Rationale (D02-RAT)
  - Background on development of OSRA
  - History of OSRA development
  - Rationale of design decisions – focus on COrDeT-3
- Example tooling
  - User Manual for Example Tooling (D04)
- OSRA Training Material
  - Not covered in COrDeT-3

Specification Role

- High-level normative specification
  - Try to specify (normatively) the minimum required
- Primarily specifies **interfaces**
- Informative material to outline design
  - Expectations
  - Intentions
- Informative material intended to be sufficient to
  - Introduce the OSRA
  - Provide a context for the specification
  - Act as a starting point for the normative document set
- The OSRA specification is not
  - A design document
  - A justification for the specification
  - An explanatory guide to the OSRA

Document Structure (1)

- Split into four main chapters
  - Based on architectural layers
- Chapter 2 is intended to be an overview
  - Provide an introduction for a reader not familiar with concepts
  - Deliberately light on detail for accessibility and brevity
- Intended to provide enough information to show the design path
  - From original user need to final architecture
- Description of SCM
  - Very high level concepts introduced to support argument
  - More detailed description of SCM in next chapter
- Process description included
  - Overview of process as a direct answer to user needs

Document Structure (2)

- Focus of Chapter 3 is the Component Layer
  - SCM is the most important part of this
- Introduce the models and their relationships
  - Core and extended models
- Detail all key concepts of the Core Model
  - Acts as an introduction and an overview for Meta-Model (SCM) Specification
  - Refer to SCM Specification and meta-model for normative specification
- Specify all pseudo-components
  - In terms similar to the training material (for consistency)
  - Discussed in the context of functional patterns (e.g. reporting, commanding etc.)
  - Illustrated with short examples
  - Pseudo-component interface specification **[Normative]**

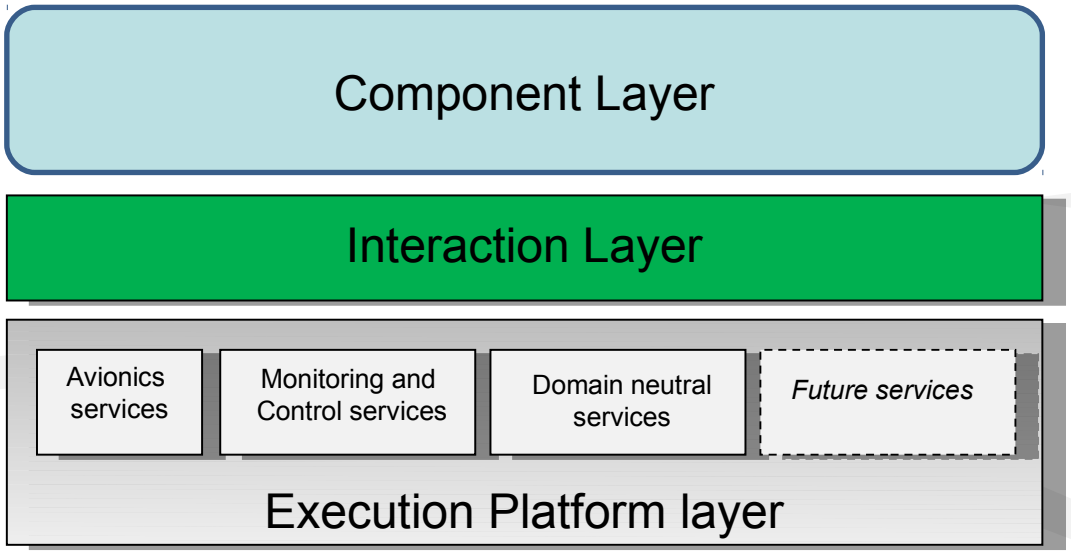
Document Structure (3)

- Focus of Chapter 4 is the Interaction Layer from the perspective of components
- Division of responsibilities
  - e.g. Where is data stored? Who is responsible for synchronisation? How does initialisation work?
  - Semantics described in text, unless specification is needed as e.g. state diagrams
- Some focus on tasking and concurrency
- Specification of component-container interface **[Normative]**
  - As abstract service primitives

Document Structure (4)

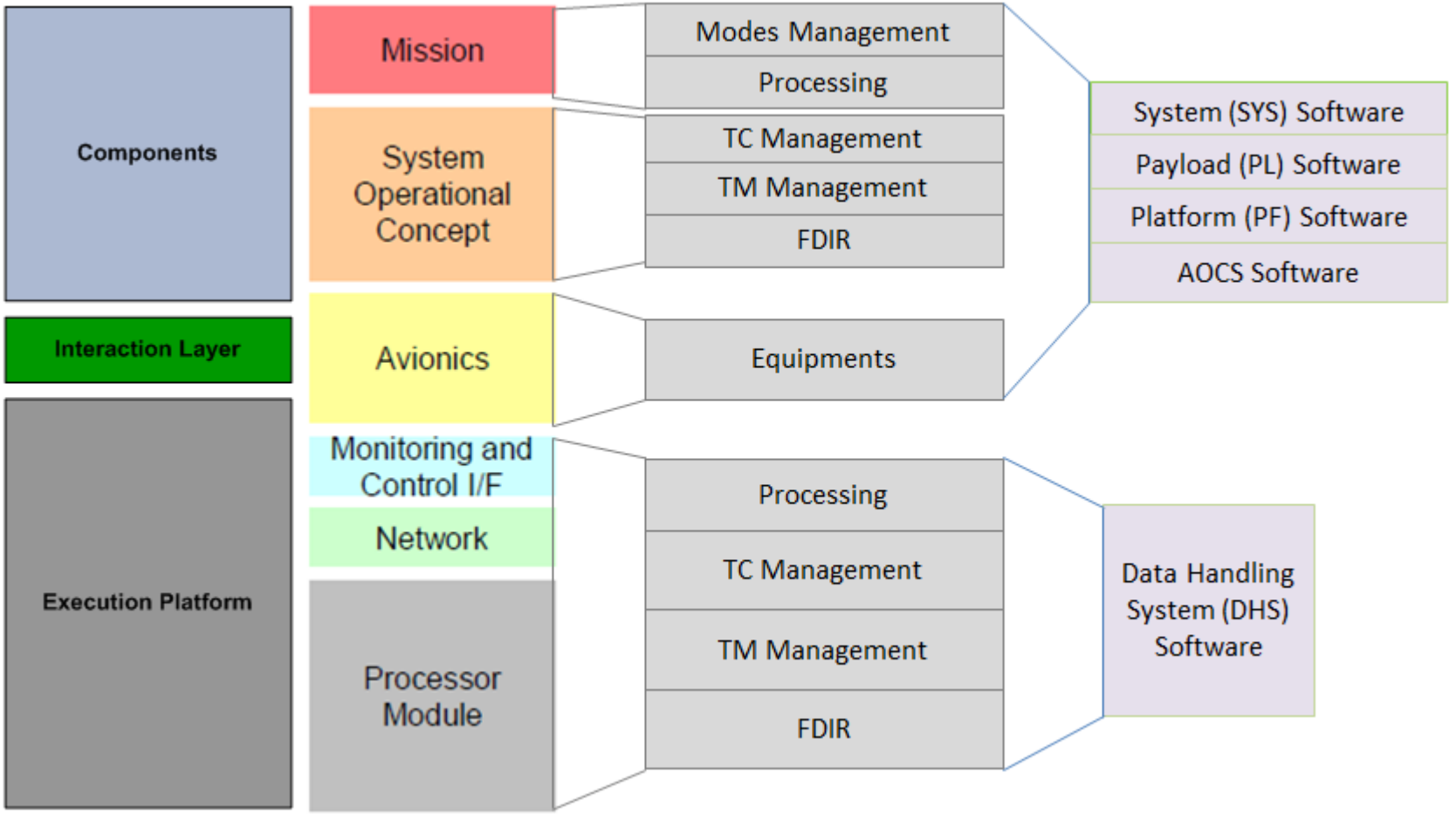
- Focus of Chapter 5 is the Execution Platform
- Covers Execution Platform services to the Interaction Layer
- Specified as abstract service primitives **[Normative]**
- Includes relationship with and use of SOIS services
  - This is also **[Normative]**

OSRA Architectural Layers



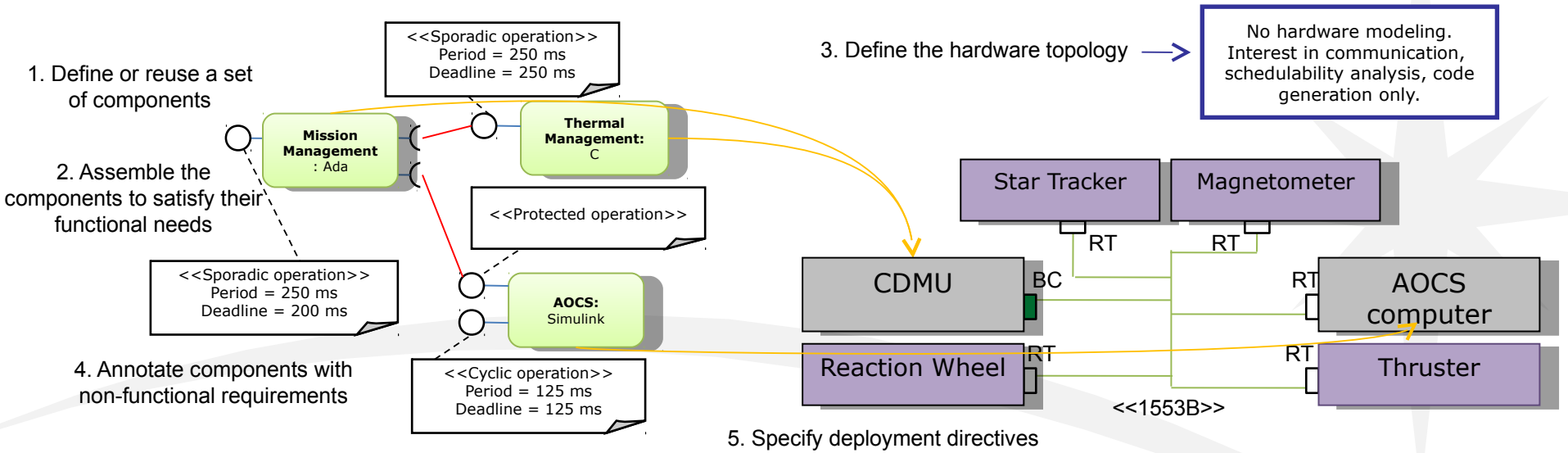
bright ascension

OSRA Layers and Variability



bright ascension

Working with Components



6. Automatically analyse the software **model** to ascertain that the **whole set** of non-functional properties can be fulfilled

bright ascension

Component Concepts

- **Interfaces**
  - Provided/required
  - Have attributes and operations
- Interface **attributes**
  - Read-only (DAT)
  - Read-write (CFG)
- Interface **operations**
  - Input, output and input/output **parameters**
  - Can throw **exceptions**
- **Events**
  - Emitters/receivers
- **Data**
  - Emitters/receivers

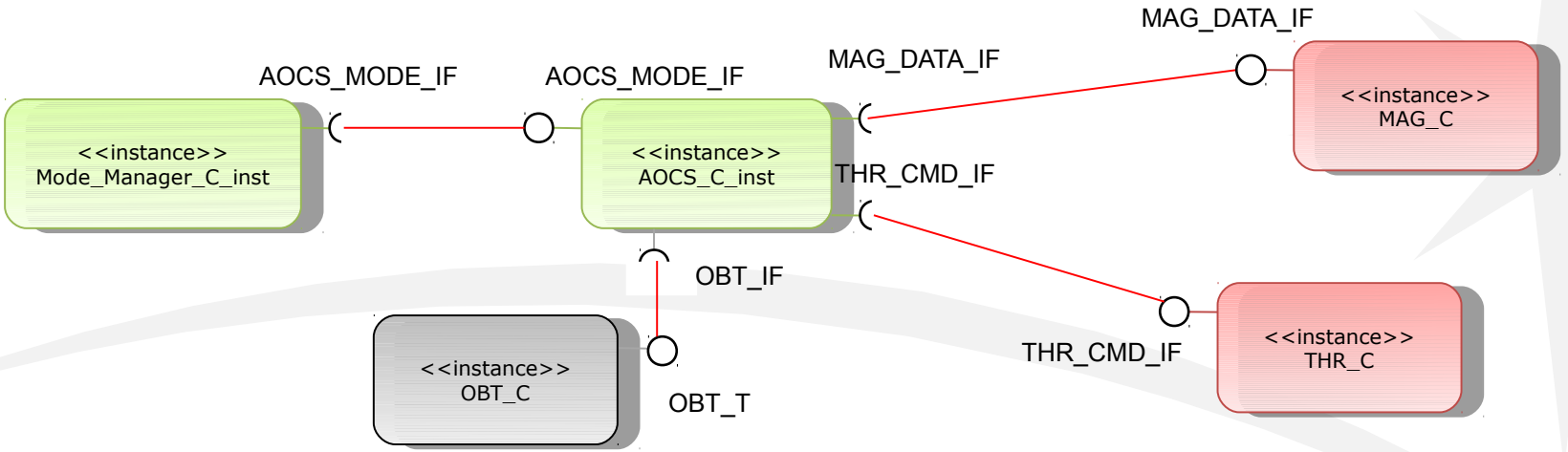
Building Software

- Component instances are assembled
- Create functional software
- Three areas remain:
  - How to do I/O (e.g. onboard)?
  - How do access platform services (e.g. time)?
  - How to do M&C
- All three involve interaction with the Execution Platform
  - How is this visible at the Component Layer?
- Use pseudo-components
  - Execution Platform services exposed with component interfaces
  - Appear as components

Device Pseudo-Components

- All onboard I/O carried out using device pseudo-components
- The provided interface(s) of the pseudo-component match the expected interface of the device
  - Expected to be at a functional level
  - Permits the later deployment of the pseudo-component onto a choice of devices
  - Separation of concerns
- Strong links with SOIS Electronic Data Sheets
- EDS specifies a provided interface which can be transformed into an SCM provided interface
  - At DVS level (more later)
- Can also make use of data emission
  - Asynchronous emission of data e.g. telemetry
  - At a functional (not packet) level

Pseudo-Component Use



bright ascension

Monitoring and Control

- From a component perspective M&C involves
  - The commanding of component operations
  - The observation (and reporting) of component attributes, events and data
  - The monitoring of component attributes
- Possible to determine which attributes and operations apply to M&C
- Attributes may be marked as **observable**
  - Can then be seen by the Execution Platform for M&C purposes
  - Includes both reporting and monitoring
- Event emitters and data emitters may also be marked as observable
- Attributes may be marked as **modifiable**
  - Permits their value to be modified by M&C
  - Only if the attribute is read-write (obviously)
- Operations may be marked as **commandable**
- All applied to **component instances**

Control over M&C

- High-level mission-specific functions are expected to be implemented at the Component Layer
- Do not need to “understand” M&C
  - Do need to **control** M&C
- M&C Services
  - Reporting
    - Enable/disable reporting of housekeeping, events etc. depending on spacecraft mode
  - Monitoring
    - Enable/disable monitoring of particular attributes
  - Commanding
  - Automation
  - Forwarding
  - OBCPs

Interaction Patterns

- Operations may also be be marked as providing acknowledgements
  - Must be marked as part of interface definition
  - Affects component implementation
- Acknowledgement types
  - None (normal operation invocation)
  - Single (acknowledgement on “acceptance”)
  - Progress (progress updates)
- Can be used throughout the software system
  - Especially applicable to M&C
  - Used to address PUS Service 1

Platform Pseudo-Components

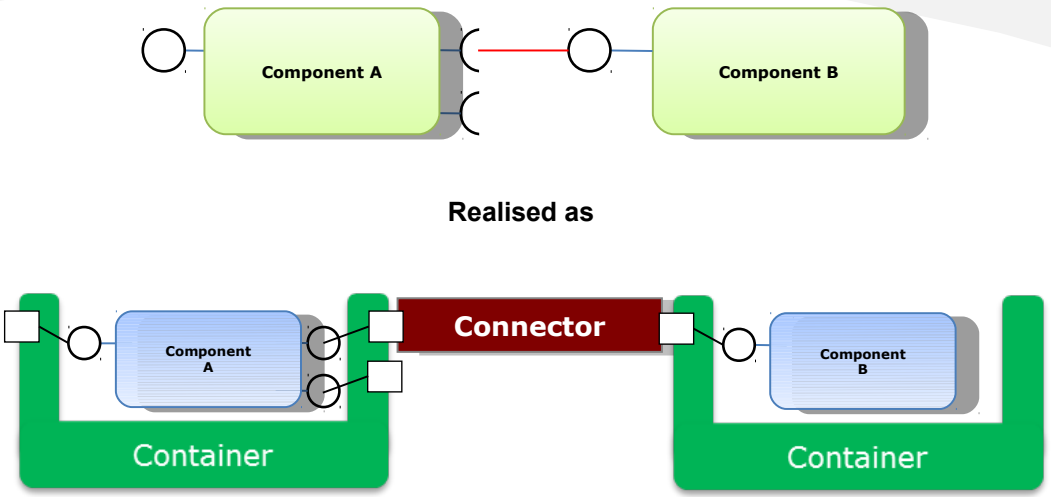
- Time access
  - In the form of “clocks”
  - Can be multiple e.g. OBET/SCET vs GPS
- Control of platform
  - Restarts
  - Reporting of errors from/to platform
- Partition control (for TSP)
- Schedule control (for TSP)

Dynamic Architecture

- Specifics of dynamic architecture not present in core model
  - e.g. assignment and control of RTOS tasks
- Semantics specified at component level
  - Sporadic (asynchronous) operation invocation
  - Periodic operation invocation
  - Execution order constraints on periodic invocations
  - Data protection
- This specifies only the non-functional requirements
  - Not the design or implementation
  - e.g. relationship between a periodically executed operation and the actual RTOS task is not defined by the core model
- Can also specify constraints on component initialisation order
  - Not necessary to specify complete initialisation order

Interaction Layer Specification

- The Interaction Layer is responsible for “glueing” the component implementation instances to the Execution Platform
- Must realise Non-Functional Properties using Execution Platform services
- Expected to be tool-generated
  - But does not have to be
- Implementation is open
  - For interoperability specify component/container interface



Component/Container Interface

- Life-cycle
  - Two-stage initialisation
  - Context management
- Provided and required interfaces
  - Attribute accessors
  - Operations
- Event emitters and receivers
- Data emitters and receivers
- No further interfaces to Execution Platform
  - No “internal API”
- Component responsible for holding attribute data
  - Complete encapsulation
  - So that implementation is flexible

Interface Specification

- Interaction Layer interface specified as **OSI service primitives**
- Container as the service provider
  - Component as the service user
- Permits the definition of semantics in a language independent way
- Needs to be bound to a specific language and invocation pattern for a concrete implementation
- Choice of bindings impact computational model
  - And vice versa

Execution Platform

- Execution Platform interface also defines as service primitives
  - Execution Platform as the service provider
- Covers all services necessary for containers and for pseudo-components
- Services
  - Reporting
  - Monitoring
  - Commanding
  - Automation
  - Forwarding
  - OBCPs
  - Platform management
  - Partition management
  - Device and Time Access
  - Life-cycle management
  - Context management
  - Tasking and concurrency

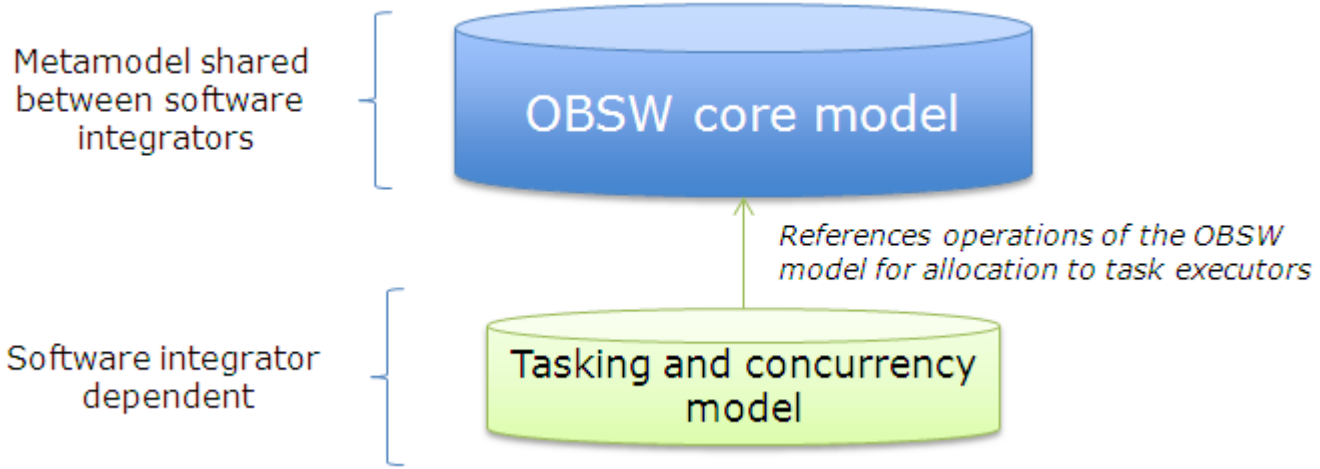
Document Status

- OSRA Specification
  - Well developed draft
  - Some known issues (e.g. too much detail on process in chapter 2)
  - Currently out for SAVOIR-FAIRE/-IMA WG review
  - Work continuing
- OSRA Rationale
  - Early draft
  - Composed of material from COrDeT-3 consortium members
  - Needs editing and more work
  - Work continuing

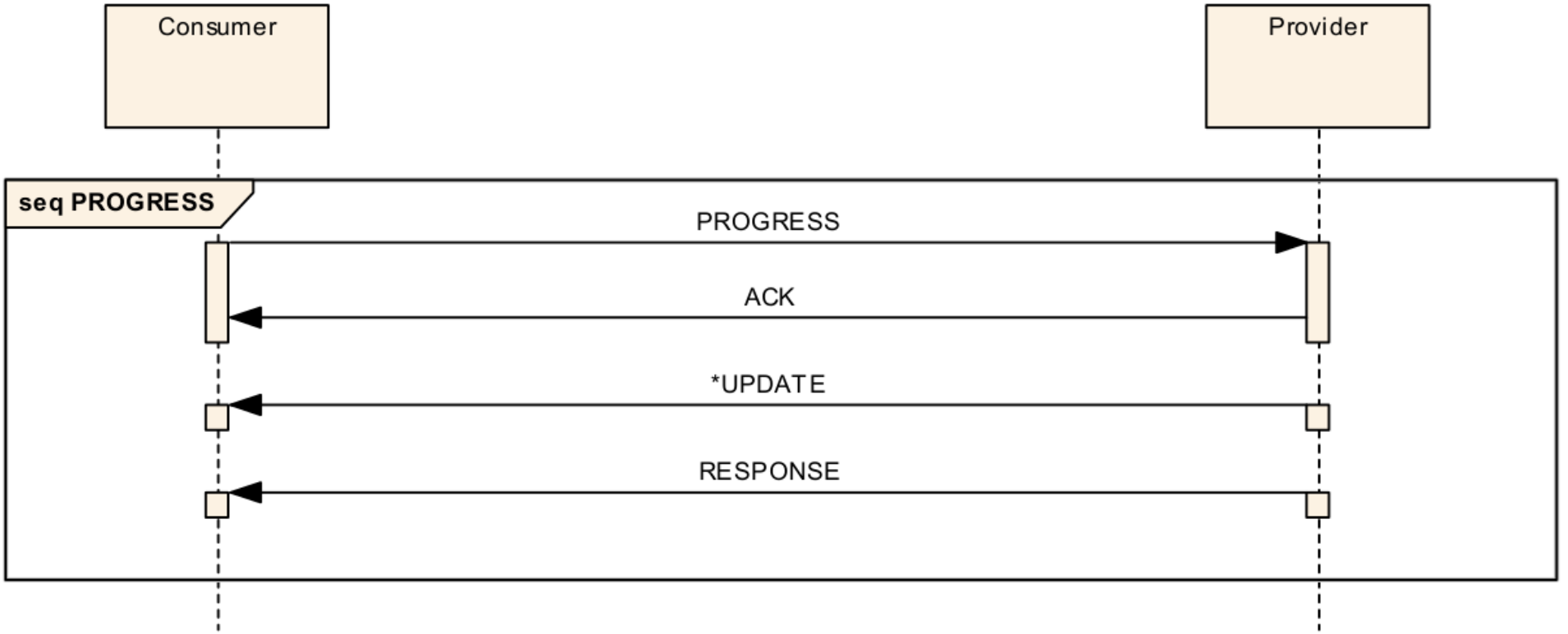
Backup Slides

Core and Extended Models

- Architecture reflects separation of concerns
- Component Layer is independent of computational model
  - Therefore SCM must also be independent of computational model
- Component Layer is independent of M&C technology
  - SCM must also be independent
- But this information is necessary for complete toolchain
- Solution is to split models into **core** and **external**
  - OSRA Specification covers core model only



Interaction Patterns (2)



bright ascension

M&C Example: Reporting

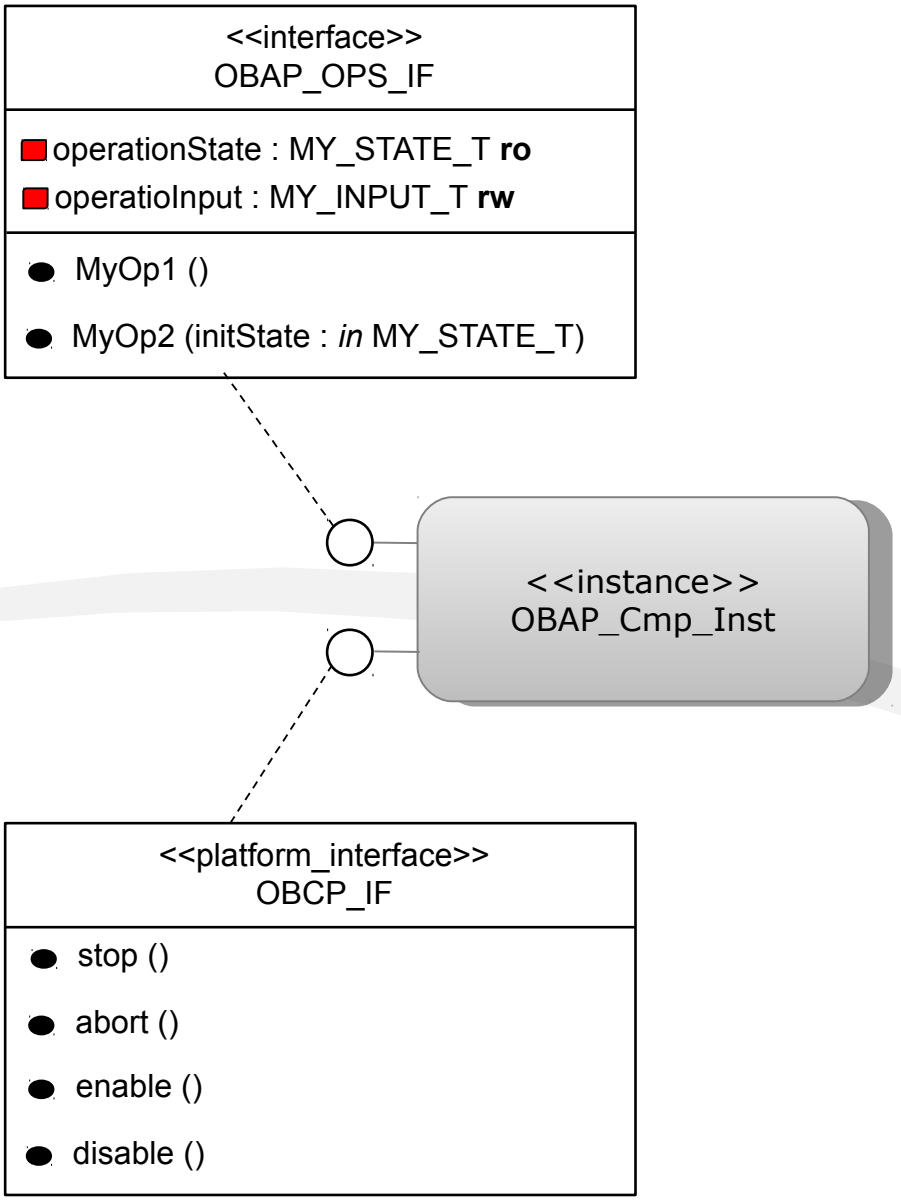
- Attribute reporting corresponds to telemetry reporting
  - e.g. housekeeping
  - e.g. PUS Service 3
- Execution Platform responsible for querying observable attributes and reporting them
- How to report them (e.g. structure definitions, telemetry packet definitions) internal to Execution Platform
  - Typically configurable
- A reporting group can be mapped to several things
  - Choice of Execution Platform
  - Not specified
- For example, a group could be mapped onto
  - A structure
  - A telemetry report packet type
  - An aggregation
- This permits components to enable/disable housekeeping packets

bright ascension

OBCPs

- Only some OBCPs are visible to component layer
  - Usually OBAPs
  - Should exist for life of software
  - Should have consistent interface for life of software
- Largely indistinguishable from other components
  - Interface to OBCP operations and attributes (parameters) using component interface
- Difference for OBCPs is that their execution may be controlled
  - Stop, abort etc.
  - Additional provided interface

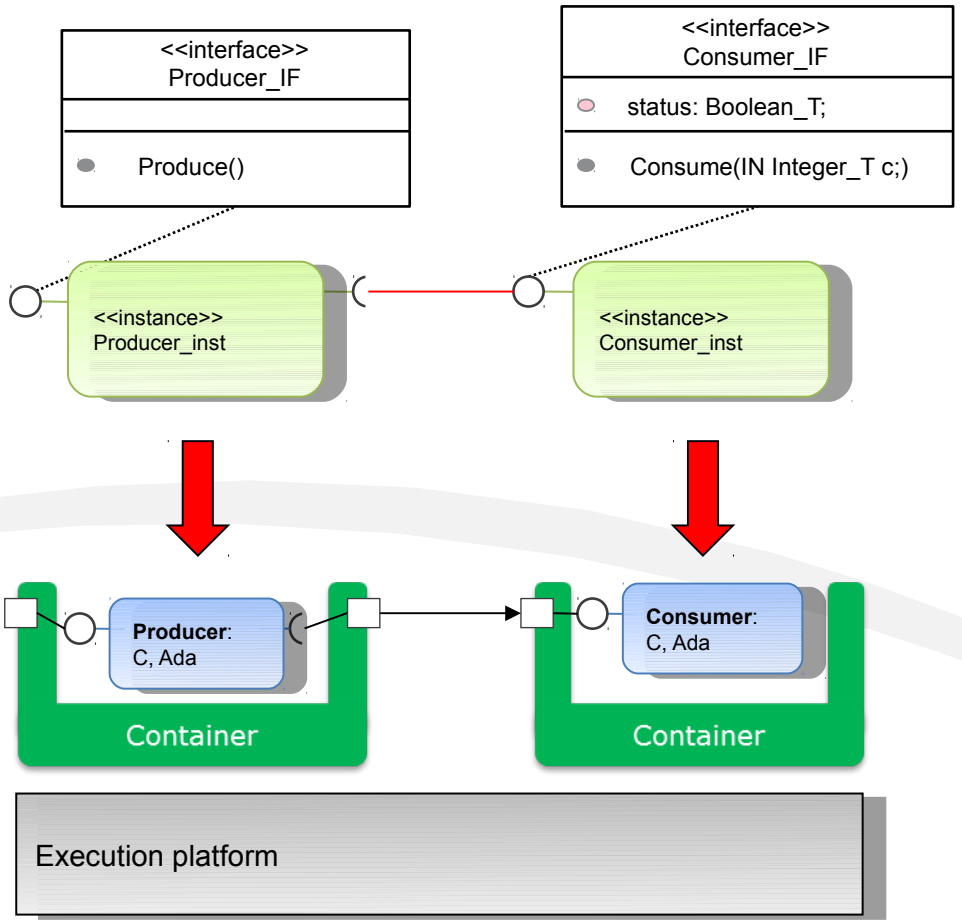
OBCP Component Interfaces



Language Bindings

- Service primitives defined in pairs
  - Request/confirmation
    - Initiated by service user
  - Indication/response
    - Initiated by service provider
- Can be bound to any(?) language
- Can be bound to a variety of patterns
  - Synchronous and asynchronous
- For example
  - Synchronous implementation of request/confirmation can become a function call

Example



bright ascension

Primitive Bindings: Producer

- Provided interface

  PI_<*Provided Interface Port Name*>_invoke<*Operation Name*>.indication

  PI_<*Provided Interface Port Name*>_invoke<*Operation Name*>.response

- Can be bound to a single, synchronous callback function (e.g. in C)

```
status_t Producer_PI_Producer_IF_invokeProduce(
   Producer_Inst_t *instance);
```

- Note that the primitives assume some kind of namespacing for component
  - Obviously not the case in C, so component type name included
- Required interface

  RI_<*Required Interface Port Name*>_invoke<*Operation Name*>.request

  RI_<*Required Interface Port Name*>_invoke<*Operation Name*>.confirmation

- Again, bound to a single, synchronous function (not callback)

```
status_t Producer_RI_Consumer_IF_invokeConsume(
   Producer_Inst_t *instance, Integer_t c);
```

Primitive Bindings: Producer

- Required interface
- Get accessors

  RI_<Required Interface Port Name>_get<Attribute Name>.request

  RI_<Required Interface Port Name>_get<Attribute Name>.confirmation
- Set accessors

  RI_<Required Interface Port Name>_set<Attribute Name>.request
- **RI_<Required Interface Port Name>_set<Attribute Name>.confirmation**
- Again, bound to single, synchronous functions

```
status_t Producer_RI_Consumer_IF_getStatus(
   Producer_Inst_t *instance, Boolean_t *status);

status_t Producer_RI_Consumer_IF_setStatus(
   Producer_Inst_t *instance, Boolean_t status);
```

Primitive Bindings: Producer

- Asynchronous bindings also possible
- One possible implementation is callbacks
  - Other possible e.g. tickets
- For example, consider Consume as asynchronous with progress reporting
- Covered by existing service primitives

  RI_<*Required Interface Port Name*>_invoke<*Operation Name*>.request

  RI_<*Required Interface Port Name*>_invoke<*Operation Name*>.confirmation
- Function to be called

```
status_t Producer_RI_Consumer_IF_invokeConsume(
   Producer_Inst_t *instance, Integer_t c);
```

- Callbacks

```
status_t Producer_RI_Consumer_IF_invokeConsumeStarted(
   Producer_Inst_t *instance, status_t status);

status_t Producer_RI_Consumer_IF_invokeConsumeProgress(
   Producer_Inst_t *instance, status_t status, Progress_t progress);

status_t Producer_RI_Consumer_IF_invokeConsumeComplete(
   Producer_Inst_t *instance, status_t status);
```

External Systems

- Interactions with external systems also carried out using Execution Platform services
- External systems could be
  - Other partitions
  - Other OBCs (e.g. payload computer)
  - Other spacecraft
  - Possibly ground
- Defined at a high level
- Messaging services not provided
- Capability to
  - Get/set a remote attribute
  - Invoke a remote operation
  - Receive a remote event
  - Etc.