The OSRA Component Model / Meta-model reference implementation

Presented by:

Marco Panunzio
marco.panunzio@thalesaleniaspace.com

ADCSS 2014
ESA-ESTEC, Noordwijk, NL
27/10/2014

OPEN

ThalesAlenia Space
A Thales / Finmeccanica Company

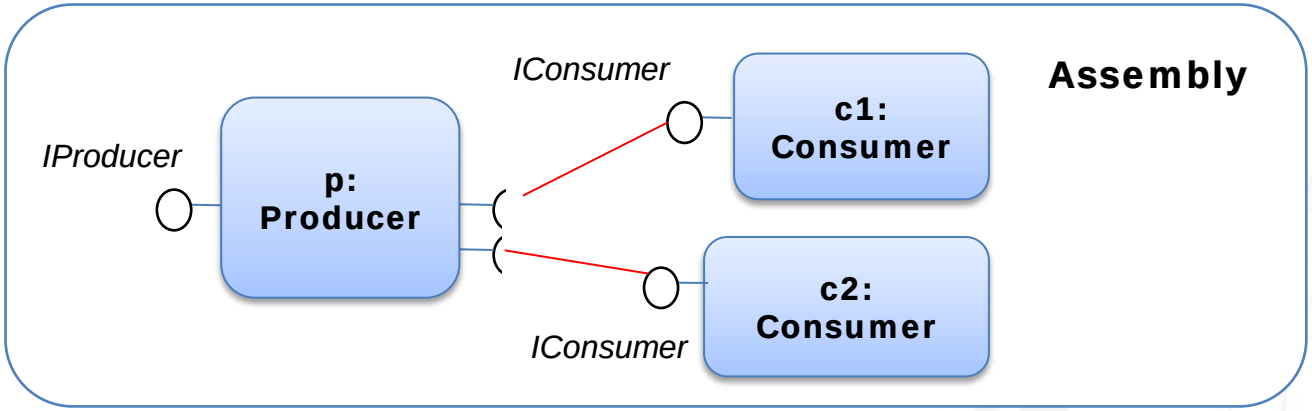
MDE and CBSE: Abstraction


```
class Producer : public IProducer {
  private:
     IConsumer* c;
  public:
    void produce() {…};
    void setC(IConsumer* _c) {…};
}

class Consumer : public IConsumer {
  void consume(Item i) {…};
}
```

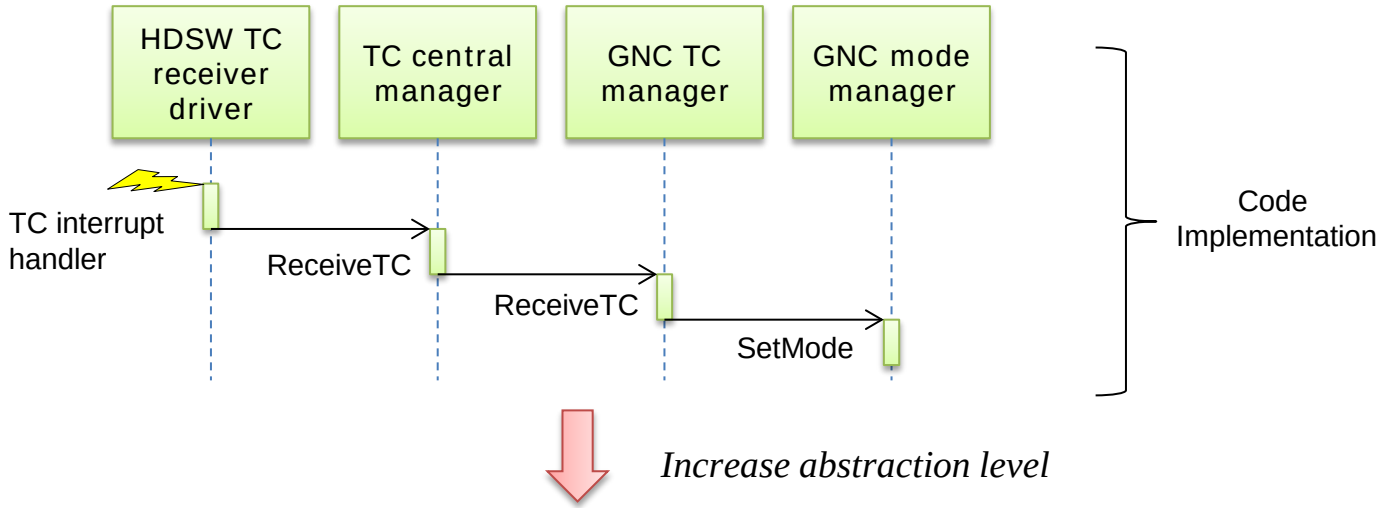
```
class Assembly {
  private:
     IProducer* p;
     IConsumer* c1;
     IConsumer* c2;
  public:
    Assembly() {
       p = new Producer();
       Consumer* cs = new Consumer [2];
       c1 = new Consumer(); cs[0] = c1;
       c2 = new Consumer(); cs[1] = c2;
        p.setC(cs);
    }
}
```

Can you spot the software design?

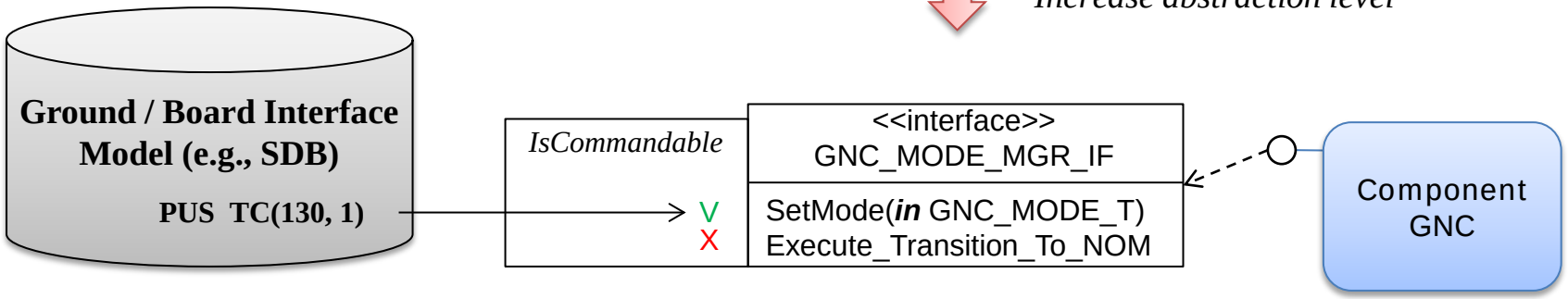


MDE and CBSE: Abstraction and code generation

Example: pattern for TC reception and dispatching



Increase abstraction level



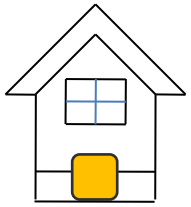
Use Model-Driven Engineering and Component-oriented development
to specify the pattern with simple design constructs.
The code implementation is automatically generated with known code patterns.

Separation of concerns

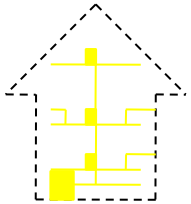
- It is about
  - Understanding the different *concerns* of the development process
    - Functional, behavior, timing, tasking, etc…
  - Clearly identifying the *actors* of the development process
    - Software architect, component developer, etc…
  - Assigning concerns to development actors
    - Establishing clear responsibilities
      - Responsibility on one concern shall not be overlapping
        - Yet several users can use and refer to the concern specification
    - Ensuring that the actor has all the relevant information for the specification of the aspects of the concern
  - Understanding *when* to address those concerns

Separation of concerns

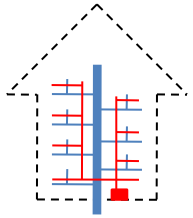
- It is enacted
  - At software modeling level
    - By using separate “design views” (a partial representation of the system focused on a single concern)
      - Enforces non-overlapping responsibility on design entities



“Structural view”



“Electrical wiring view”

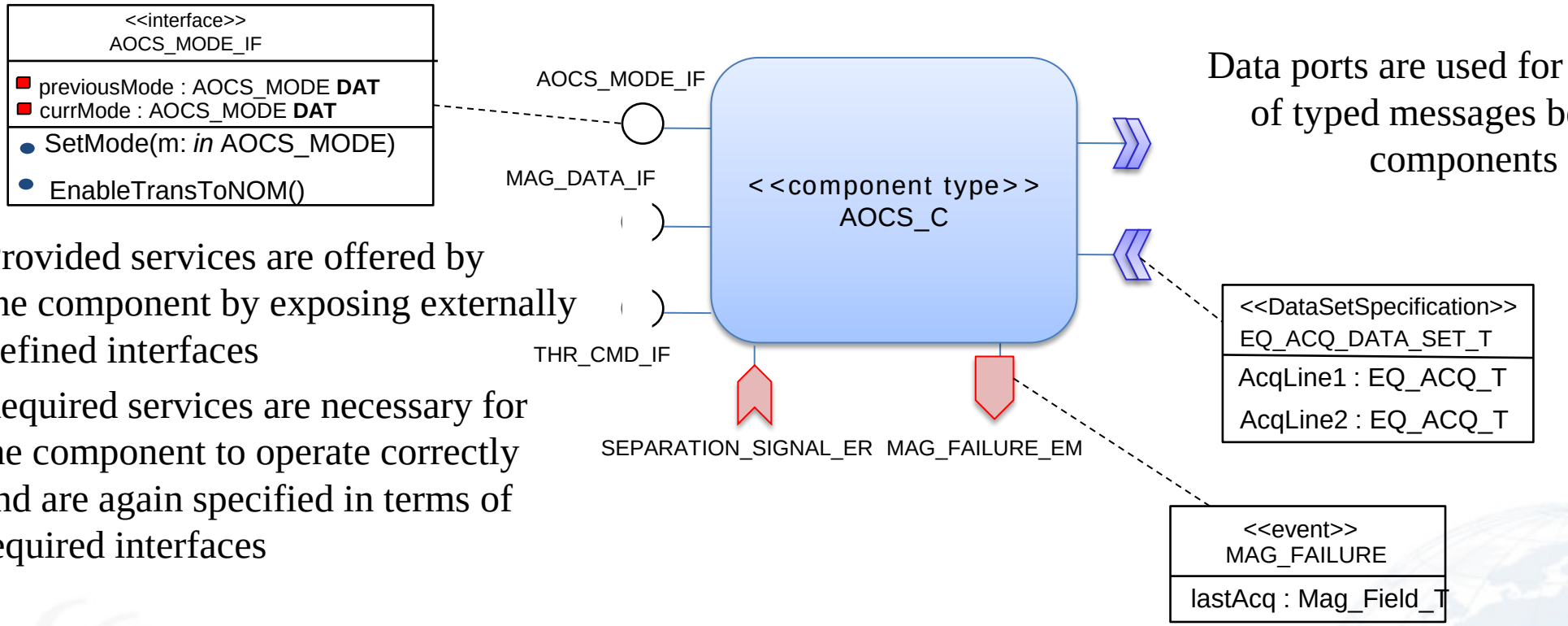


“Plumbing view”

  - At software implementation level
    - By using dedicated software entities so as to decouple and address separately relevant concerns
      - E.g., functional concerns separated from interaction and communication concerns; functional concerns separated from tasking concerns

Component type

A **component type** can be considered as a **specification** defining the boundaries of the component, the services it will offer to external clients and the services it needs from other components in order to operate correctly.



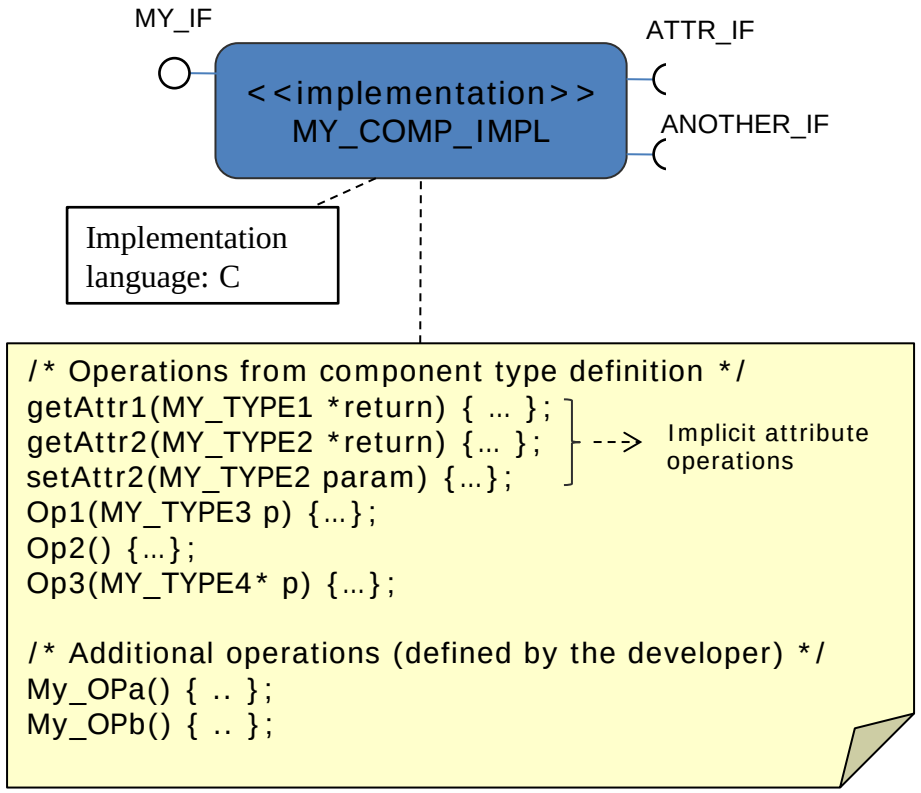
Data ports are used for exchange of typed messages between components

Provided services are offered by the component by exposing externally defined interfaces

Required services are necessary for the component to operate correctly and are again specified in terms of required interfaces

Emitted and Received Events
Events are asynchronous notifications to communicate that something notable has happened (e.g., state change, failures, etc…)

Component implementation



A component implementation is a concrete implementation of a component type.

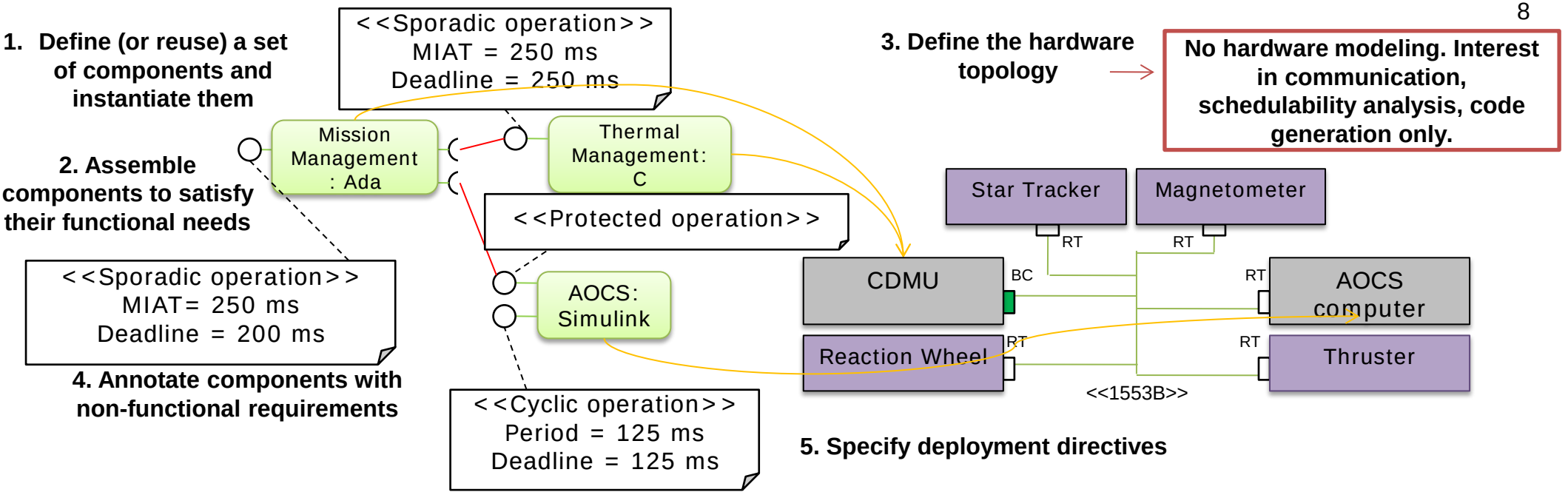
It is the subcontracting unit of the approach. Its realization can be delegated to an external stakeholder.

It may be subject to detailed design (internal decomposition, additional internal operations…)

"Resource and implementation requirements" (e.g., max WCET, max EEPROM / RAM size) can be stipulated on the implementation prior to delegating its realization.
The component developer shall develop it honoring the stipulated budgets.

The component developer develops the source code (pure **sequential code**) of the services of the component developed in the programming or modeling language of choice (C, Ada, Simulink, etc…).

Instantiation, deployment and non-functional properties



- Non-functional properties are only declaratively specified
  - Their implementation is assured by suitable containers
    - Dependent on the chosen execution platform and computational model
  - The set of non-functional properties is fixed and decided by the OSRA methodology

Access to devices and execution platform services

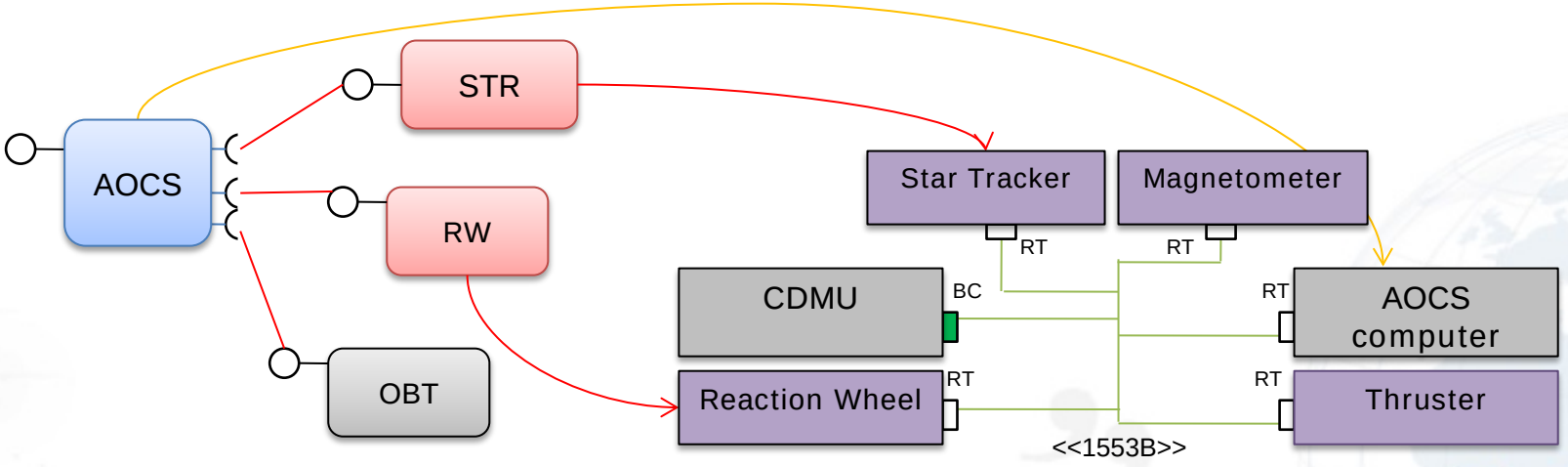
Take advantage of the high-abstraction level of the design to declare in the same way access to software components, avionics equipment and services of the execution platform.

Avionics equipment

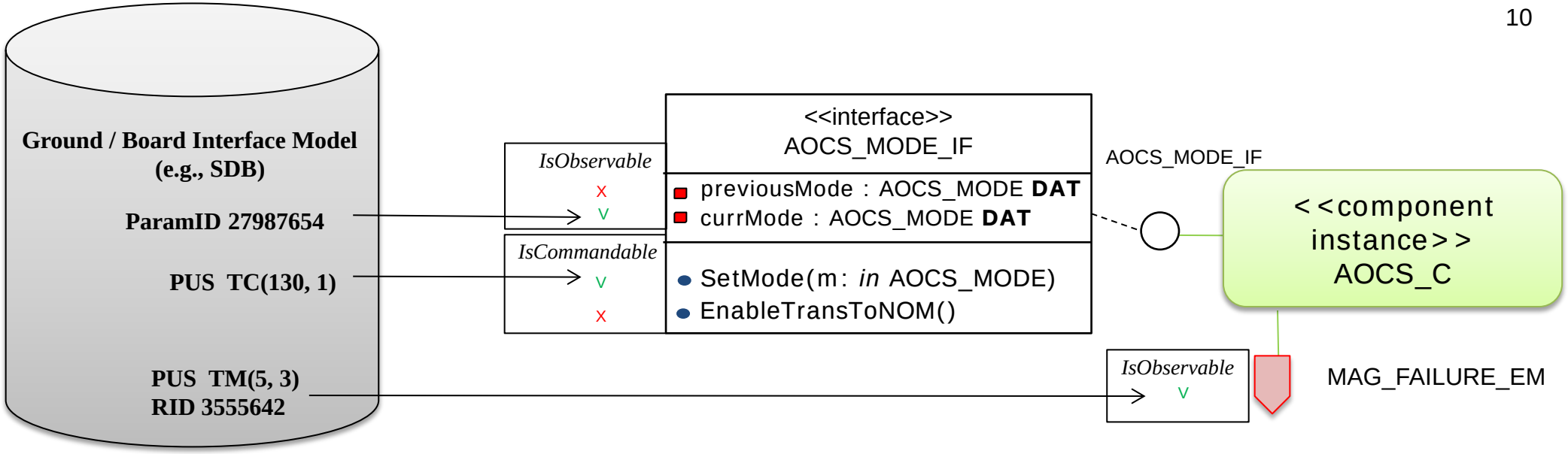
1. Declare that the AOCS component requires data from the Star Trackers and sends commands to the Reaction Wheels
2. Deploy the equipment “pseudo-component” on its hardware counterpart
3. Generate the communication code to the equipment
   - Leveraging also on the communication support services (e.g., SOIS) of the execution platform

Execution platform service

1. Declare the access to the OBT provided by the execution platform
   - Using a “pseudo-component” even if the execution platform is not developed with components

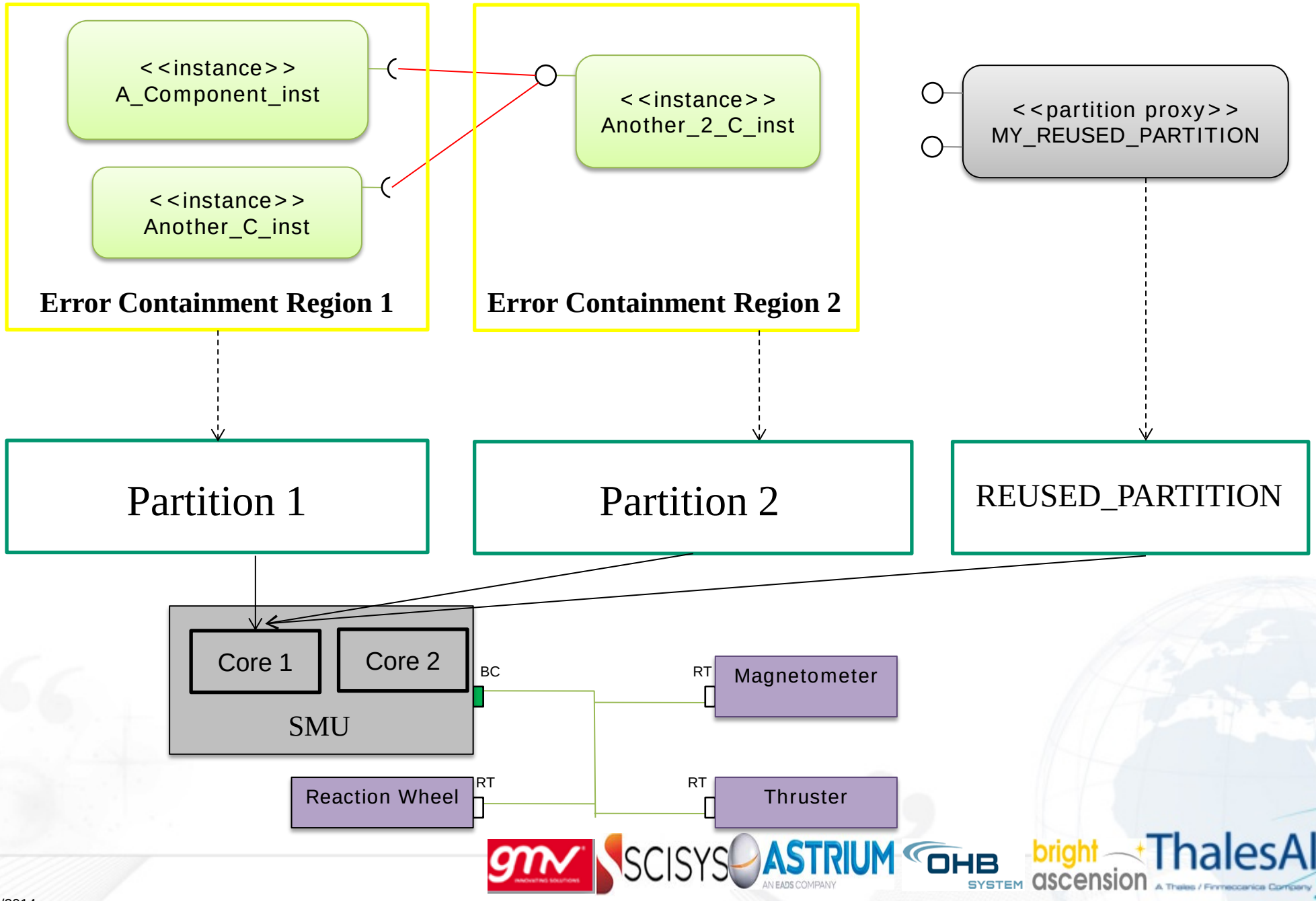


Commandability and observability



- Declare which operations, interface attributes and events are commandable / observable on the ground / board interface
- They can be referenced by an “external model” (e.g., an SDB) in order to provide all complementary information required for their exploitation (e.g., ParamIDs, mapping to PUS TC and PUS params, etc…)

Error containment regions and partitions

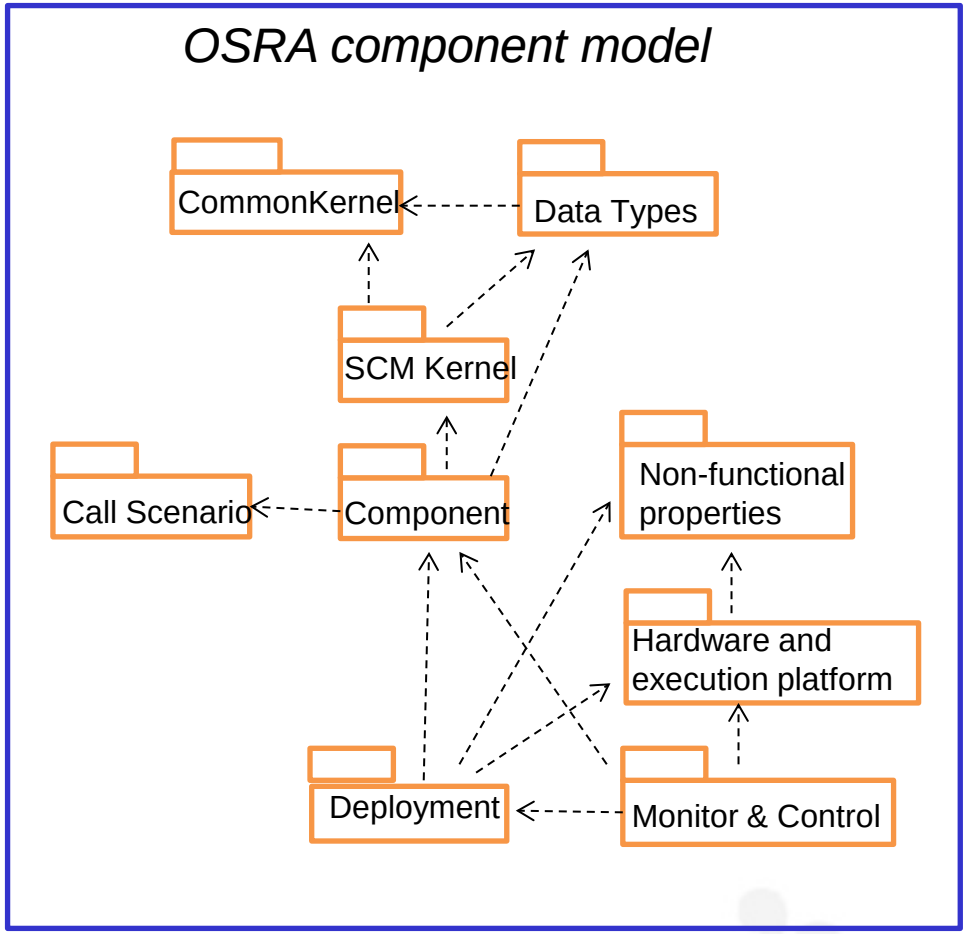


OSRA core model and external models

- OSRA core model (i.e., included in the component model)
  - Functional description of services
  - Component instantiation, deployment and declarative specification of NFPs
  - Functional relationships with devices, execution platform services, Monitor & Control services
- OSRA “external models”
  - HK plan, protocol-dependent information (PUS-A, PUS-C, MOS)
  - Operation allocation to tasks and semaphores
  - Execution platform configuration
  - TSP partition configuration

OSRA metamodel specification: language architecture

- The component model language is organized as a set of “language units” comprising a cohesive set of language concepts



OSRA metamodel reference implementation

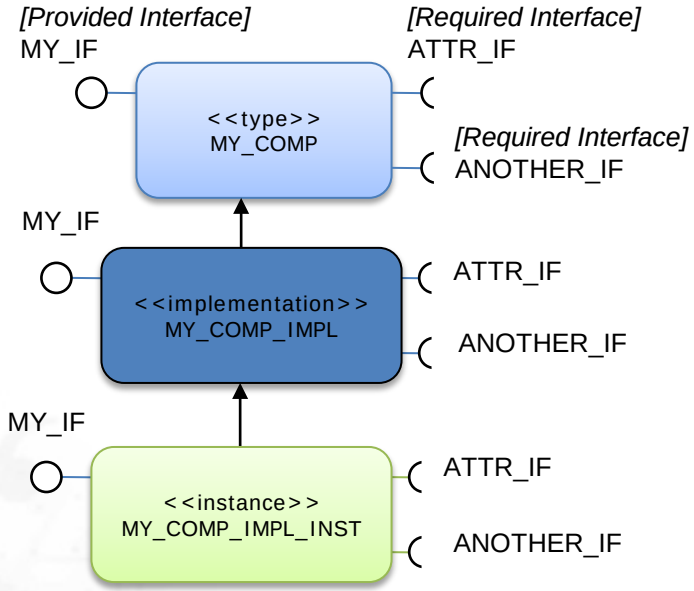
- Implemented as a domain-specific metamodel (DSM) called SCM (“Space Component Model”)
  - All entities of the component mode are first-class native entities of the language
    - Clear and direct mapping of concepts from spec to metamodel
  - Avoid offering concepts too generic or open to interpretation
    - Foster unambiguous understanding of concepts and convergence by the community towards the same “reference concepts”
  - Offer as much as possible a design space with admissible choices only
    - Less “a-posteriori” checks on the model are necessary to ensure consistency of design
    - Less backtrack of design due to incorrect choices
- In principle only a possible implementation of the component model
  - Other implementations can be possible, as long as *syntax* and *semantics* relationship between the component model and the implementation language is ensured

End of presentation

Questions?

Overview of the OSRA component model

- How do we define components?
  - We define three different entities: component types, component implementation and component instances
    - In accord with the separation of concerns principle, each one of them represent different concerns and has a specific role in the software development process



A **component type** can be considered as a **specification** defining the boundaries of the component, the services it will offer to external clients and the services it needs from other components in order to operate correctly.

A **component implementation** is one concrete implementation of a component type. The component implementation entity created in the OBSW model has the same external interfaces of its component type.
It can be subject to detailed design (to add internal decomposition and additional operations).
The component developer will code the **implementation code** for the component implementation

A **component instance** is an instantiation of a component implementation. Component instances are important for: (i) **binding components** to fulfill their functional needs; (ii) **deploying components** on processing units; (iii) specify the desired **concurrency, real-time and M&C requirements**.

OSRA metamodel reference implementation

CommonKernel

Metaclasses for basic needs:

- Unique identifiers
- Named elements
- Elements with comments and
- Annotations

Reusable for various modeling studies, independently of the OSRA

CommonTypes

Definition of data types and constants.
Expressive power comparable to strongly typed languages (e.g., Ada)

Reusable for various SW- or OSRA-related studies (e.g., SOIS EDS?)

SCM

Complete metamodel for the definition of an OSRA “core model”