Ada Ravenscar Code Archetypes for Component-based Development

Marco Panunzio[1] — Thales Alenia Space – France
Tullio Vardanega — University of Padova, Italy

17th International Conference on Reliable Software Technologies – Ada Europe 2012

Stockholm, June 12, 2012

[1]work performed while at the University of Padova

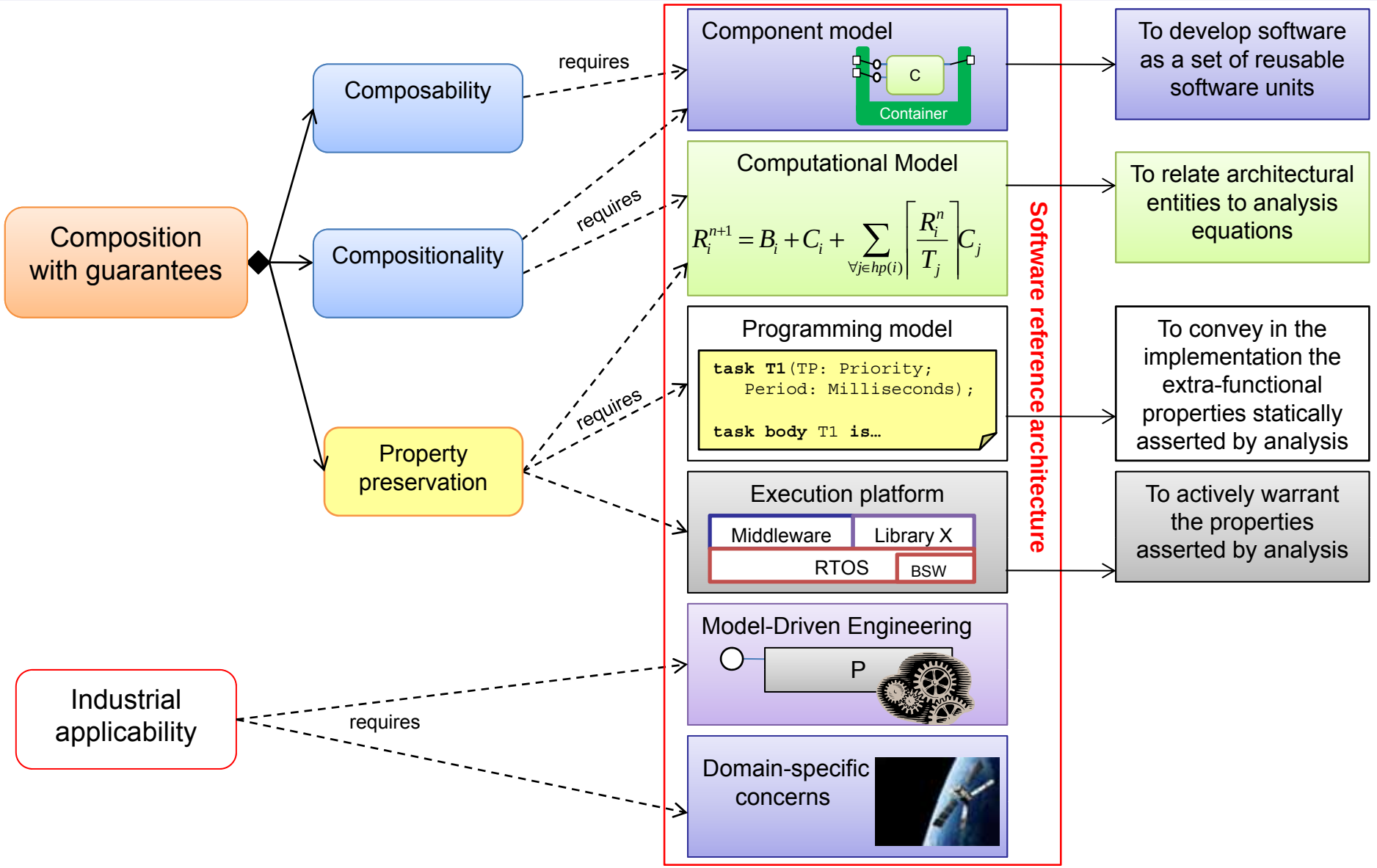
Outline

- Introduction
  - Rationale for code archetypes
  - Target context: a software reference architecture
- Founding principles
  - Separation of concerns
    - Realization in CBSE
- Component model
  - Overview
- Code archetypes
  - Example (model and source code level)
  - Containers
  - Delegation chain for extra-functional properties enforcement
  - Sporadic operations
- Conclusions and future work

Ravenscar code archetypes for CBD

- Goals
  - Support separation of concerns
    - In particular between functional and extra-functional aspects
      - Sequential algorithmic code separate from tasking, synchronization and interaction code
    - Advantages
      - Reuse of functional (algorithmic) code under different extra-functional requirements
  - Complement a defined component-oriented approach
    - We briefly outline the overall context and the component model
  - Cover the complete infrastructural code
    - Interfaces, components, communication code
    - Implementation of extra-functional properties
      - Tasking, synchronization, real-time
  - Support the inclusion of business code written in Ada or C/C++
- Leveraging on lessons learned from past R&D projects

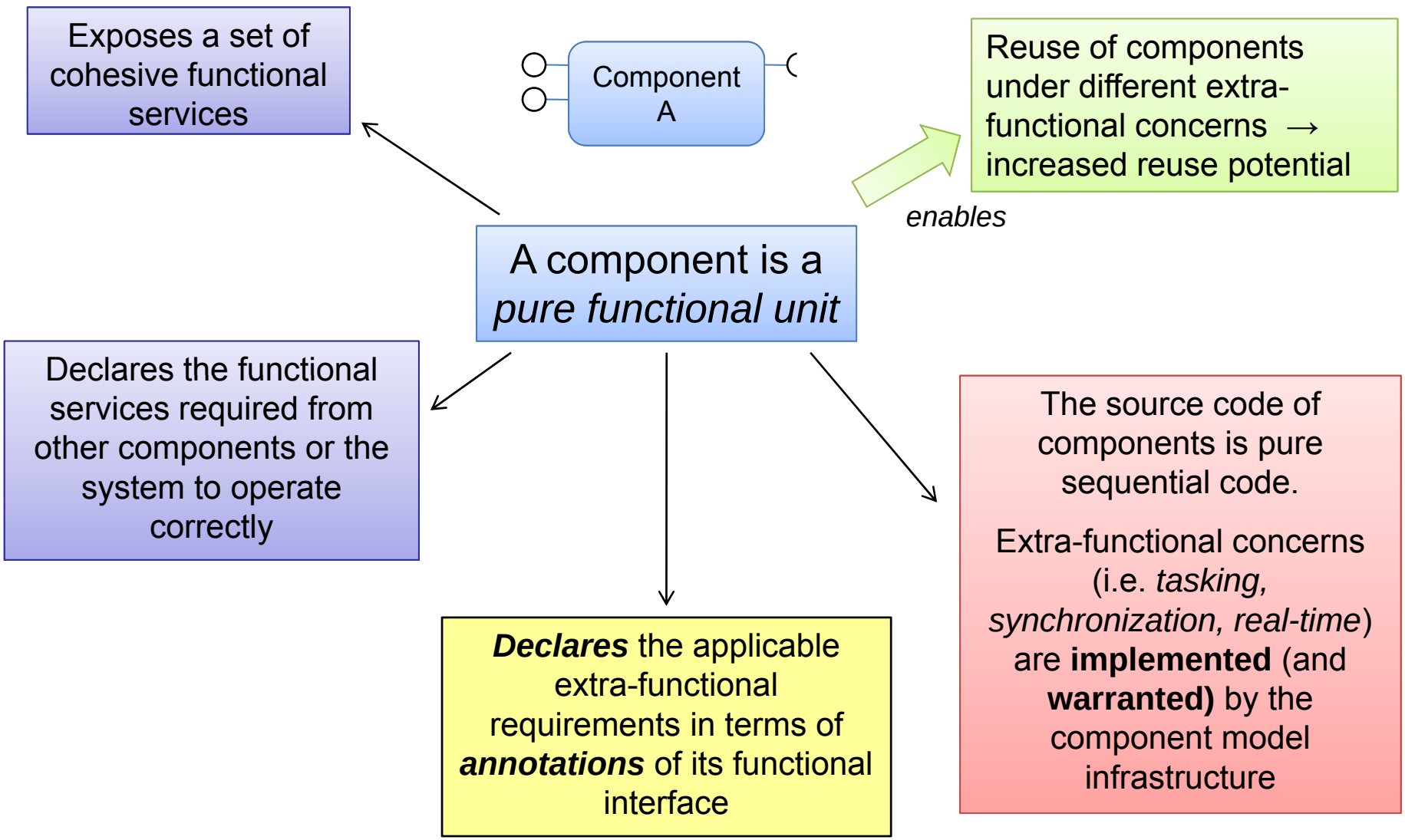
Context of use: a software reference architecture



Cornerstone principle: separation of concerns

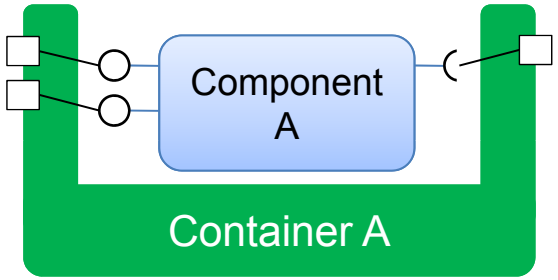
- To sharply separate different aspects of software design
  - In particular functional and extra-functional concerns
  - Allows each development actor to focus exclusively on their area of expertise
  - Fosters the use for each concern of the best-fit formalisms, tools and verification techniques
- Achieved
  - In the design space by use of design views
  - In the component model and implementation by allocation of different concerns to different software entities
    - Component, container, connector

Separation of concerns: realization in CBSE (I)



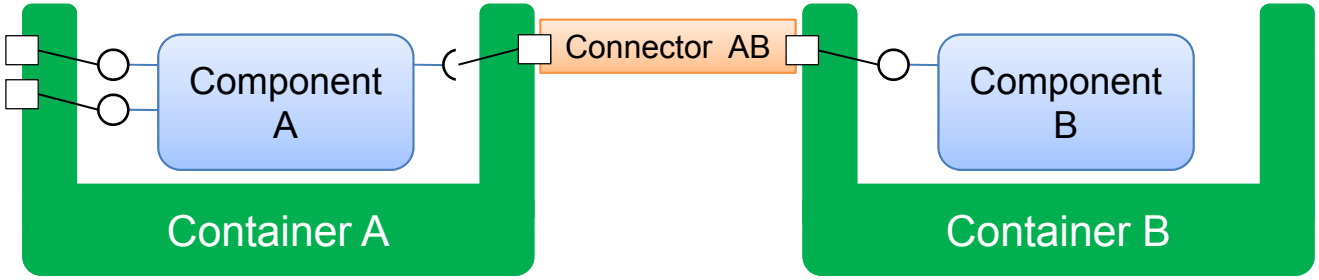
Separation of concerns: realization in CBSE (II)

Container



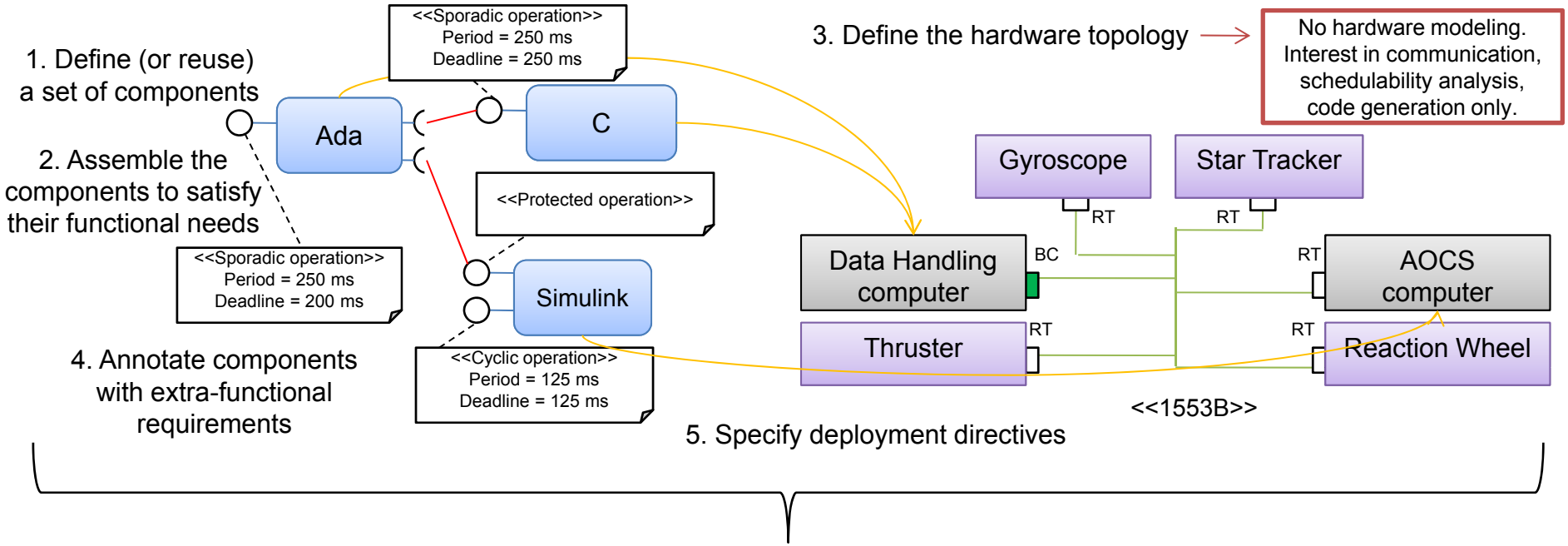
- Wrapper responsible for the declared extra-functional attributes
- Provides the component with a mediated connection with the execution platform and the system in general

Connector



- Addresses interaction concerns
- Decouples the component from the other end-point(s) of a communication
- Realizes connection properties (best-effort, at most once, exactly once)
- E.g. procedure/function call, remote message passing, I/O file operation, …

The concept in a nutshell (I)



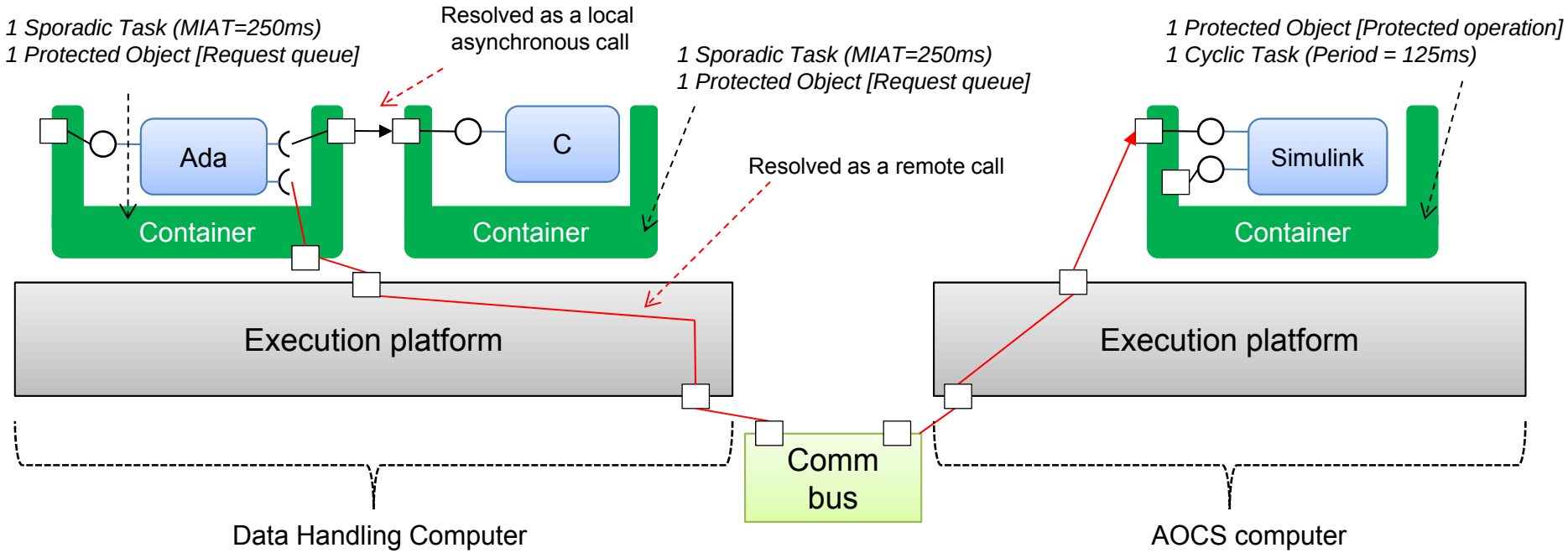
6. Automatically analyze the software **model** to ascertain that the **whole set** of extra-functional attributes can be fulfilled

Operation kind	Dedicated thread executor	Code executed by the caller	Short description
Cyclic	Y		Executed with a defined period
Sporadic	Y		Minimum separation between 2 subsequent executions (MIAT)
Protected		Y	Concurrent execution guarded with mutual exclusion
Unprotected		Y	No synchronization protocol

The concept in a nutshell (II)

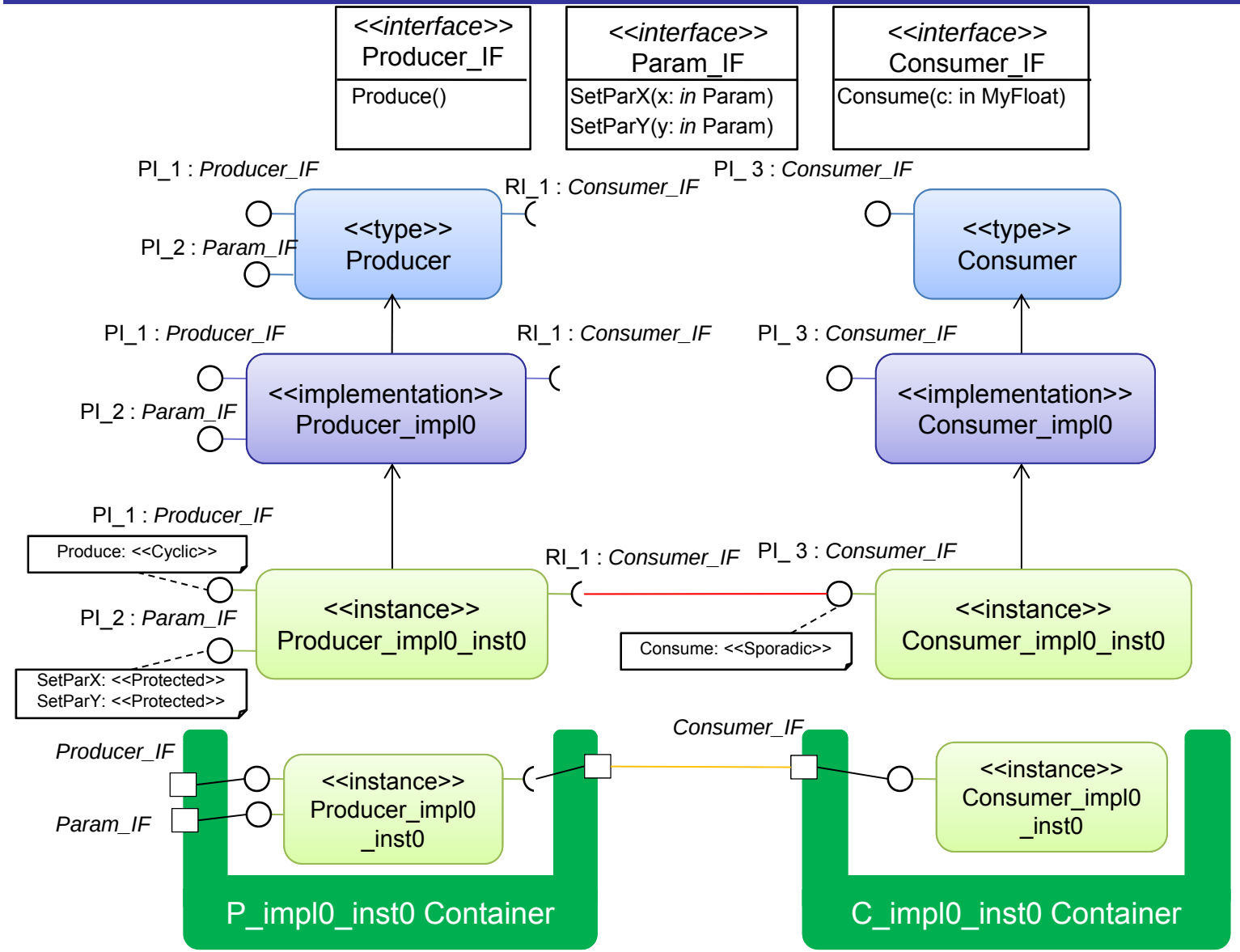
a. A model-to-code transformation wraps a container around a component so as to realize all the declared extra-functional properties (tasking, period, etc...)

b. A model-to-code transformation, using the component bindings and the deployment information, realizes the desired communication, possibly relying on the execution platform for remote communication

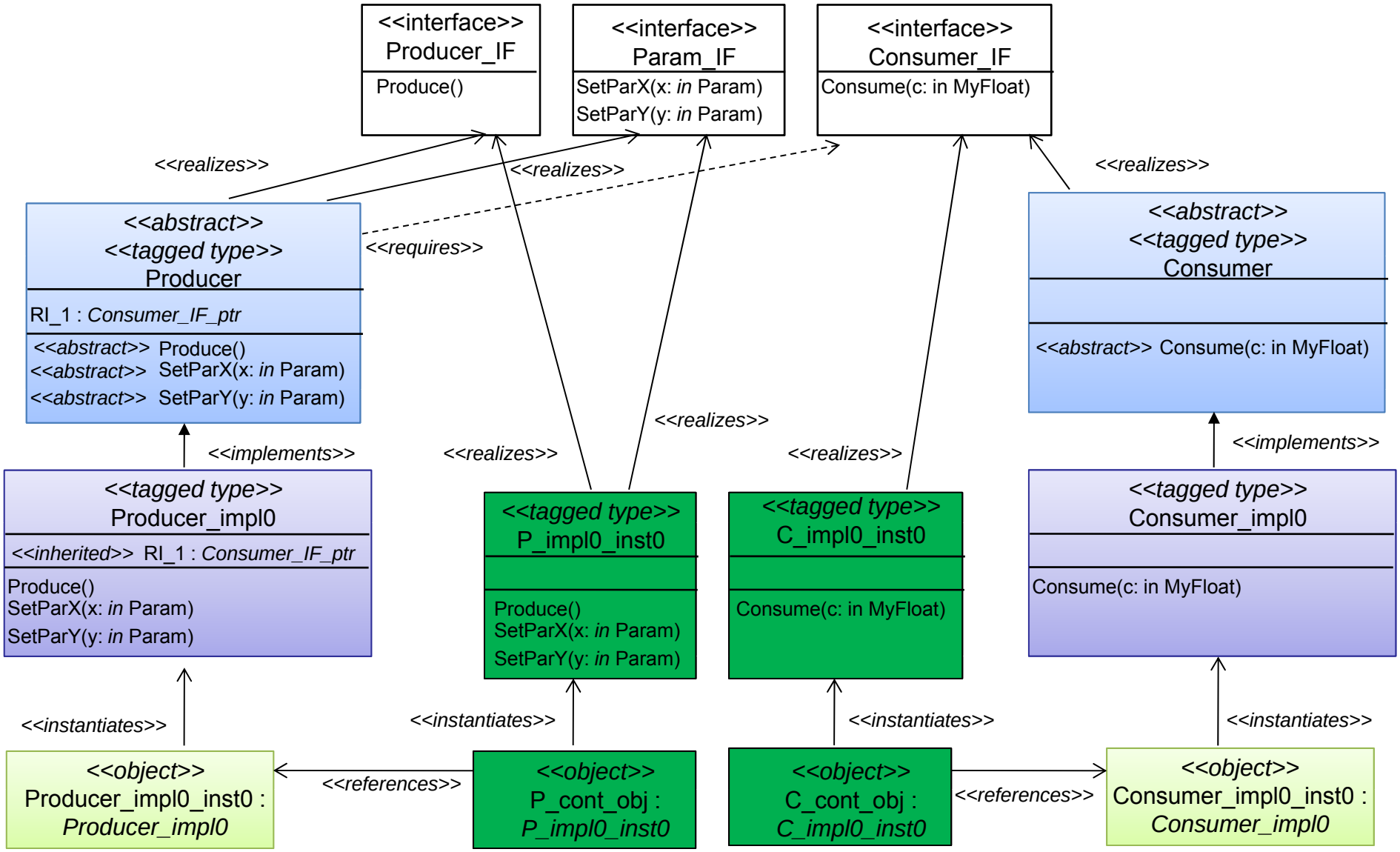


In the following we are going to present the code archetypes for the implementation of interfaces, components and containers that makes this possible.

Example (component model level)



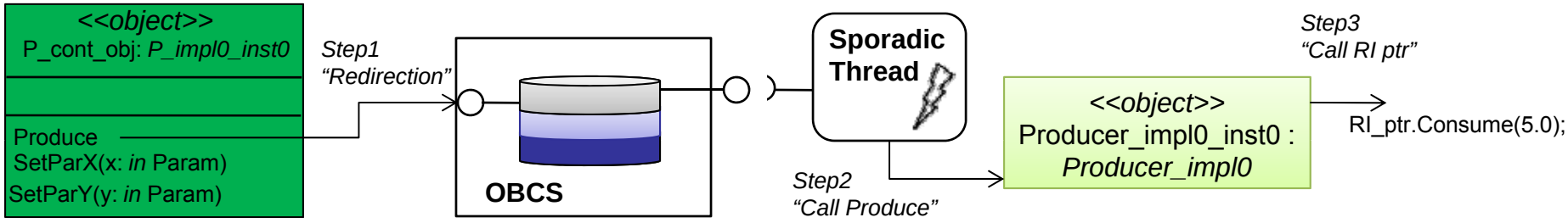
Example (source code level)



More on containers (I)

- Call to a PI operation of a container
  - Step1: Enforcement of the extra-functional properties by the container
    - Calls to sporadic and protected operations are redirected to the container entities enforcing the desired concurrent nature
    - Cyclic operations are triggered by the execution platform
    - Unprotected operations do not require special treatment [go to next step]
  - Step2: Call the corresponding operation on the component instance referenced by the container
  - [Step3]: Calls to an RI are performed on the RI pointer defined on the component type

Example: Produce tagged as <<sporadic>>

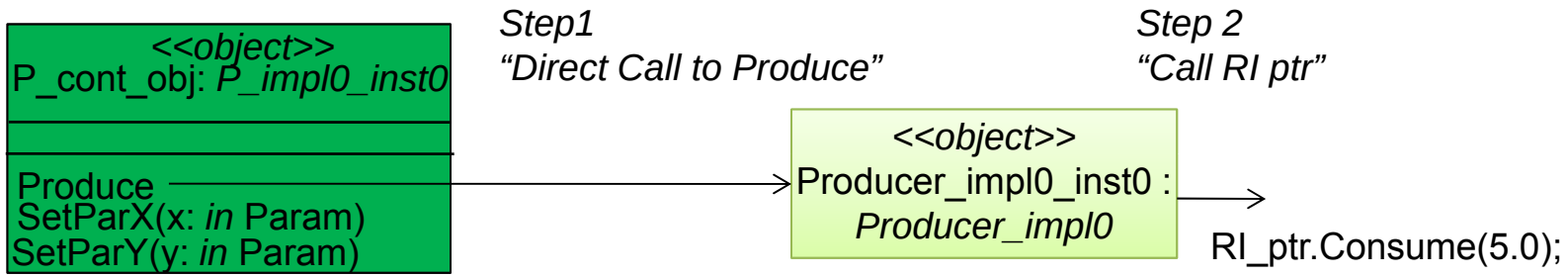


More on containers (II)

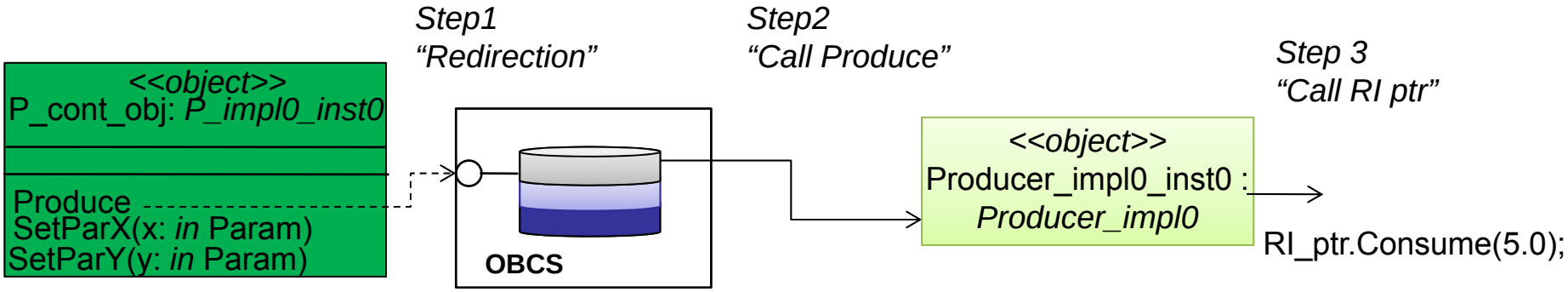
- RI pointers are set by the deployment at initialization time
  - Set with a reference to the container “encompassing” the container instance to be called
    - Possible as the container realizes all the interfaces provided by its component instance
    - So that the delegation chain for extra-functional properties enforcement is executed
- Explicit connectors are necessary to support distribution transparence
  - The connector redirects the call to the communication middleware
  - The connector realizes the interface of the RI it is connected to

Delegation chains for extra-functional properties (I)

Produce tagged as <<unprotected>>

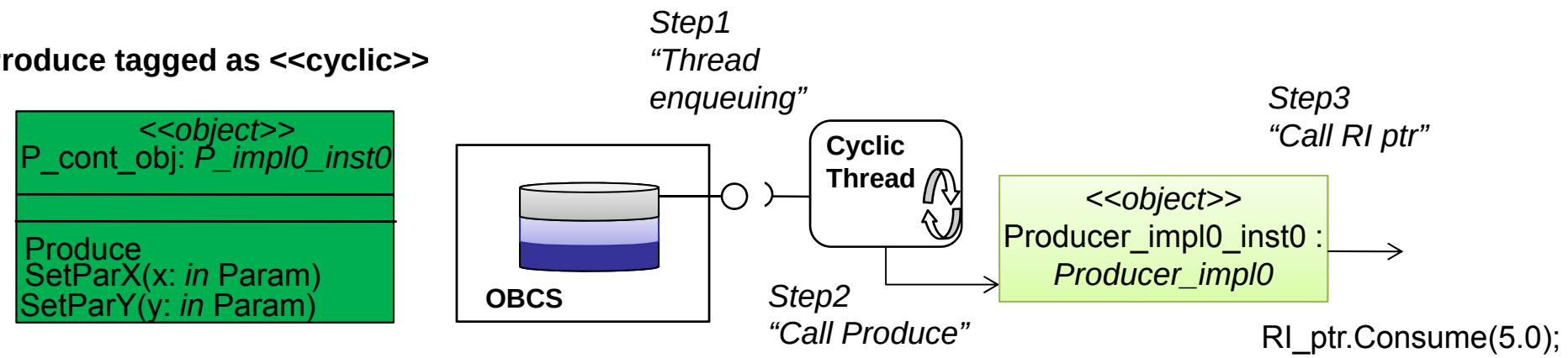


Produce tagged as <<protected>>

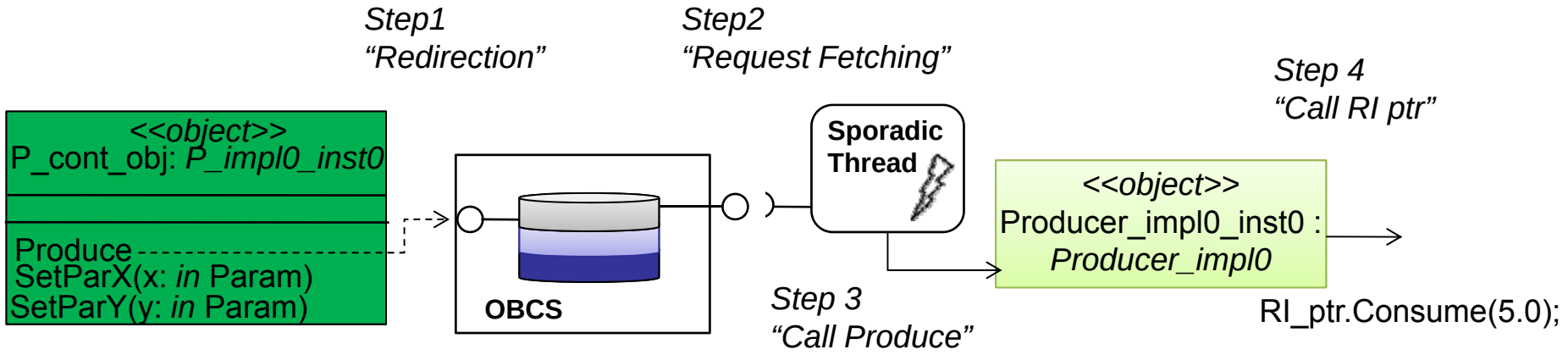


Delegation chains for extra-functional properties (II)

Produce tagged as <<cyclic>>



Produce tagged as <<sporadic>>



Sporadic Task

```
with System; with Ada.Real_Time; with Data_Structure;
package Sporadic_Task is
  task type Thread_T (Thread_Priority : System.Any_Priority;
                      MIAT : Natural;
                      Get_Request : access procedure (Req : out Data_Structure.Request_Descriptor_T;
                                                      Release : out Ada.Real_Time.Time)) is
  pragma Priority (Thread_Priority);
  end Thread_T;
end Sporadic_Task;

with System_Time; with Ada.Real_Time; use Ada.Real_Time;
package body Sporadic_Task is
  task body Thread_T is
    Req_Desc : Data_Structure.Request_Descriptor_T;
    Release : Time;
    Task_MIAT : constant Time_Span := Milliseconds(MIAT);
    Next_Time : Time := System_Time.System_Start_Time + System_Time.Task_Activation_Delay;
   begin
    loop
      delay until Next_Time;
      Get_Request(Req_Desc, Release);
      Data_Structure.My_OPCS(Req_Desc.Params.all);
      Next_Time := Release + Task_MIAT;
   end loop;
  end Thread_T;
end Sporadic_Task;
```

MIAT enforcement

Enqueue in the OBCS.
Fetch the next request or block on empty queue.

Execute the functional code of the sporadic operation (see next slide)

Calculate next wake-up time.

Sporadic operations: parameter passing

Define an abstract root for all parameter types [and its Class-wide access type]

<<abstract tagged record>>
Param_Type

procedure *My_OPCS (Self :* ***in out*** *Param_Type)* ***is abstract****;*

Define a request descriptor to reify incoming requests

<<record>>
Request_Descriptor_T
Params: Param_Type_Ref

Operation to be called:
procedure *Op1(a:* ***in*** *T1; b:* ***in*** *T2);*

Define in the container the record for the operation's parameters

```
type Op1_Param_T is new Param_Type with record
   OPCS_Instance : <<Access to the interface comprising the operation>>
   a: T1;
   b: T2;
end record ;
```

Define the **container** as a realization of the interface comprising the operation *(so that it exposes the same subprograms)*

Set the RI of client components with an access to the container

Calls by clients to **Op1** are encoded in a **Request_Descriptor_T** containing an **Op1_Param_T** and enqueued at the OBCS

Override *My_OPCS* so that it calls the functional code

```
procedure My_OPCS( Self : in out Op1_Param_T ) is
begin
 Self.OPCS_Instance.Op1(Self.a, Self.b) ;
end My_OPCS;
```

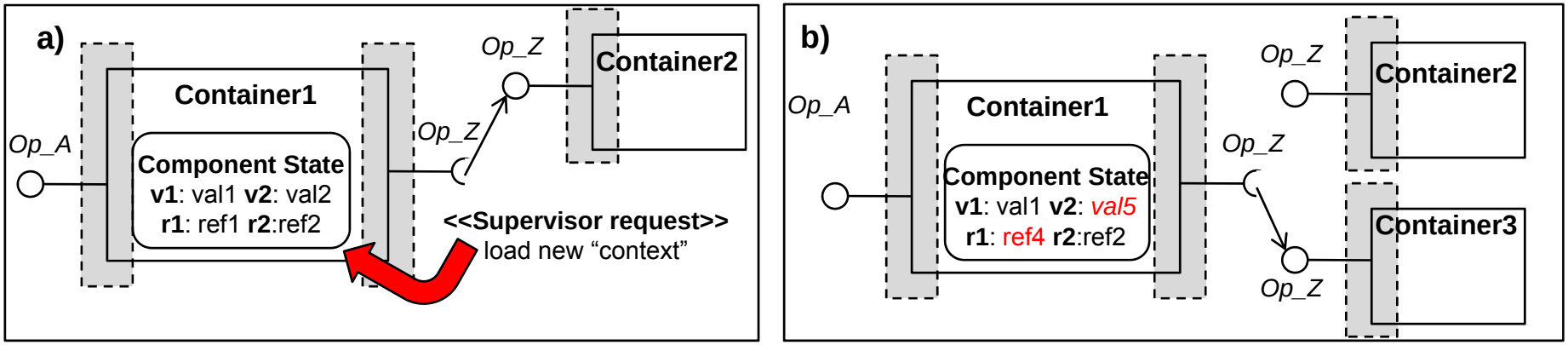
The call My_OPCS(Req_Desc.Params.all); on the request descriptor fetched by the sporadic thread from the OBCS dispatches to the functional code of *Op1* (defined in the **component implementation**)

Conclusions

- We presented a set of Ravenscar-compliant code archetypes for Ada 2005
  - Support separation of concerns
    - In particular between functional and extra-functional aspects
      - Separate sequential algorithmic code from tasking, synchronization and interaction code
    - Advantages
      - Reuse of functional (algorithmic) code under different extra-functional requirements
  - Complement a defined component-oriented approach
  - Support the inclusion of business code written in Ada or C/C++

Future work

- Management of the component state
  - The collection of the internal members (variables/parameters) of the task and the actual bindings of its required interfaces
  - Data-level protection of the component state
  - Saving, restoration and loading of it shall be possible by commanding by an external “supervisor authority”



- Support for space-specific concerns
  - At design and implementation level

End of presentation

Questions?