Date: July 2016

OBJECT MANAGEMENT GROUP®

Unified Component Model for Distributed, Real-Time And Embedded Systems

Version 1.0

OMG Document Number ptc/16-07-04

Normative reference: http://www.omg.org/spec/UCM/1.0

Associated Normative Machine Consumable Files:

http://www.omg.org/spec/UCM/20160501/ucm_contracts.ecore
http://www.omg.org/spec/UCM/20160501/ucm_datatypes.ecore
http://www.omg.org/spec/UCM/20160501/ucm_commons.ecore
http://www.omg.org/spec/UCM/20160501/ucm_interactions.ecore
http://www.omg.org/spec/UCM/20160501/ucm_technicalpolicies.ecore
http://www.omg.org/spec/UCM/20160501/ucm_components.ecore

http://www.omg.org/spec/UCM/20160501/ucm_contracts.xsd
http://www.omg.org/spec/UCM/20160501/ucm_datatypes.xsd
http://www.omg.org/spec/UCM/20160501/ucm_commons.xsd
http://www.omg.org/spec/UCM/20160501/ucm_interactions.xsd
http://www.omg.org/spec/UCM/20160501/ucm_technicalpolicies.xsd
http://www.omg.org/spec/UCM/20160501/ucm_components.xsd

http://www.omg.org/spec/UCM/20160501/core.ucm
http://www.omg.org/spec/UCM/20160501/timer.ucm
http://www.omg.org/spec/UCM/20160501/execution.ucm
http://www.omg.org/spec/UCM/20160501/interactions.ucm
http://www.omg.org/spec/UCM/20160501/properties.ucm

http://www.omg.org/spec/UCM/20160501/core.ucm.xml
http://www.omg.org/spec/UCM/20160501/timer.ucm.xml
http://www.omg.org/spec/UCM/20160501/execution.ucm.xml
http://www.omg.org/spec/UCM/20160501/interactions.ucm.xml
http://www.omg.org/spec/UCM/20160501/properties.ucm.xml

http://www.omg.org/spec/UCM/20160501/core.idl
http://www.omg.org/spec/UCM/20160501/timer.idl
http://www.omg.org/spec/UCM/20160501/execution_policies.idl
http://www.omg.org/spec/UCM/20160501/interactions.idl
http://www.omg.org/spec/UCM/20160501/standard_properties.idl

http://www.omg.org/spec/UCM/20160501/ucm.idl

This OMG document replaces the submission document (mars/16-05-04). It is an OMG Adopted Beta specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be entered by April 1, 2017 using the Issue Reporting Form on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

The FTF Recommendation and Report for this specification will be published on June 15, 2017. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

Copyright © 2016 Thales, PrismTech
Copyright © 2016 Object Management Group, Inc.

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software – Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: http://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

Table of Figures

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters

109 Highland Avenue

Needham, MA 02494

USA

Tel: +1-781-444-0404

Fax: +1-781-444-0320

Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman/Liberation Serif - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://issues.omg.org/issues/create-new-issue

1. Specification Outline

1.1 Software architectures made of components

The Unified Component Model (UCM) enables the design of software applications based on the use of components. Software applications are designed as a set of interconnected components. These components typically correspond to the application business logic of a target solution. Components interact with each other through connectors. They can also be associated with technical elements (named technical policies) that control their execution or provide services.

From the descriptions of the components with their associated connectors and technical policies, software code is organized in blocks to maintain separation between the business logic (the component body) and the technical part (the fragments). Fragments control the component body and rely on underlying execution and communication libraries. Thus, the business logic is isolated from the execution platform and can be ported or redeployed onto other platforms. Figure 1 illustrates this transformation.

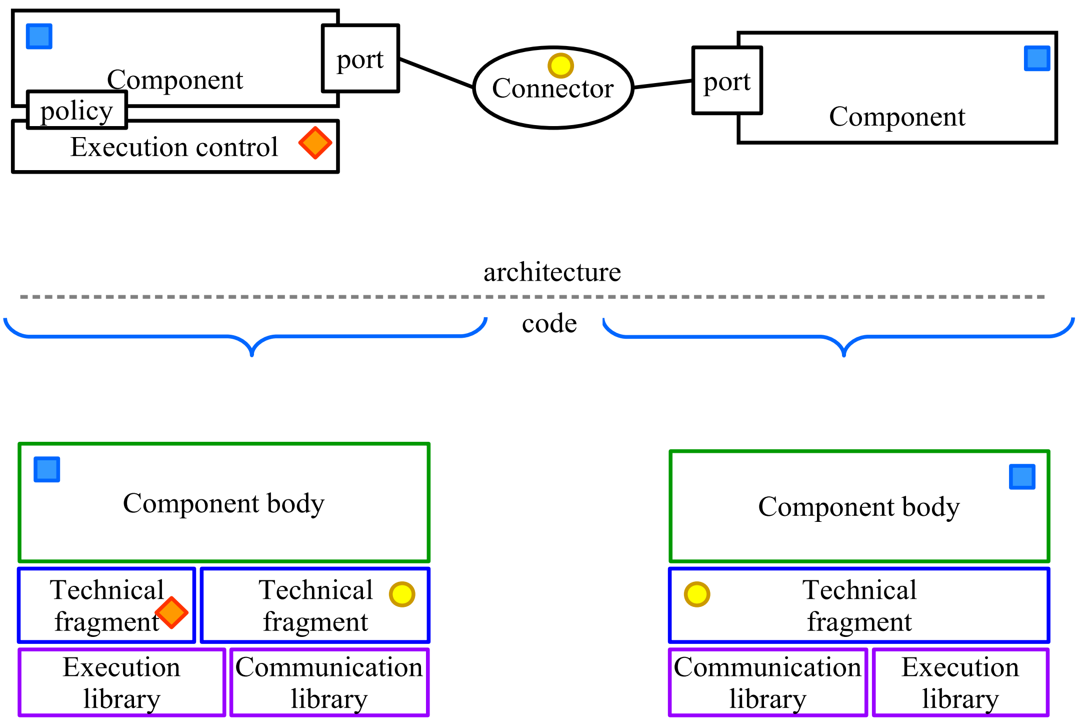

Figure 1: From components to software

1.2 A component model to design portable real-time embedded software

Design processes for real-time and embedded software systems usually have to address two opposing needs: firstly, to enable code reuse and portability, and secondly, to support domain-specific execution and communication infrastructures. UCM addresses both of these needs.

UCM consists of three main concepts: **components**, **connectors** and **technical policies**. Components represent the application business logic. Connectors implement the interaction infrastructure. Technical policies provide the execution infrastructure. Connectors and technical policies correspond to the execution platform capabilities. From an architecture point of view, they are libraries that can be used by the components, just like a programming language standard library can be used by developer code.

UCM defines a set of standard connectors and technical policies with simple APIs and semantics to ensure minimal component code portability. UCM also allows for the definition of additional connectors and technical policies to address domain-specific needs without requiring the definition of any new concepts. Such definitions can address both

API and nonfunctional parameters, such as FIFO size, priorities, etc. UCM thus supports the definition of domain-specific (or possibly cross-domain) platforms that enable component portability.

1.3 UCM actors

There are five main roles identified for component-based application engineering:

- UCM framework provider;
- UCM platform provider;
- component designer;
- component developer;
- software architect.

The *UCM framework provider* typically implements a tool set that is able to host and execute UCM components, connectors and technical policies. A UCM framework is considered to be the backbone of every UCM application.

The *UCM platform provider* defines connectors and technical policies, and provides the corresponding implementation code for a given UCM framework. UCM is designed to support extensibility by enabling the definition of additional platform elements (connectors and technical policies); several different vendors may define such platform elements. Portability of platform elements across different frameworks is not mandatory: vendors may develop framework-independent or framework-specific platform elements.

The *component designer* defines functional contracts and components, possibly complemented with nonfunctional information or requirements. Components are specified with ports corresponding to connector contracts, and are associated with the necessary technical policies.

Based on the components designed by the component designer, a *component developer* will be able to write business code that implements the functional features of a component and fulfills the component contracts.

The *software architect* defines the architecture of a particular domain application. He or she specifies one or more applications as an assembly of UCM components that rely on given UCM platforms.

These five roles can be classified into two categories: the framework provider and the platform provider **provide** the infrastructure; the component designer, the component developer and the software architect **use** the infrastructure.

A typical UCM design process may have several steps. It starts from the functional decomposition of the system into high-level software components. Then, these high-level components can be refined if needed, and decomposed into subcomponents. Component decomposition ultimately leads to leaf components that represent actual code, managed by the UCM infrastructure. Hence, leaf components are defined from the initial functional concerns, driven by the non-functional constraints, especially real-time ones (synchronization constraints, potential parallelism, etc.).

This proposal offers a hierarchical model that permits the definition of high-level components and leaf components with the same language. In the following chapters, leaf components will be called **atomic components**; components that can be decomposed will be called **composite components**.

1.4 UCM programming model

The programming model of a UCM component relies on the principle of decoupling the business code from the platform code. Only atomic components correspond to business logic; composite components are simple boxes that nest subcomponents.

The component business part and the platform parts are managed by an entity called the **container**. A container is the entity responsible for combining the business code written by the component developer with the infrastructure code provided by the UCM framework provider. Its role includes enforcing the behavior specified by the software architect in the specification of the components.

Containers will be capable of being generated automatically by the tooling that is associated to the target platform implementation. The descriptions of the component and their associated platform elements provide enough information to support this process.

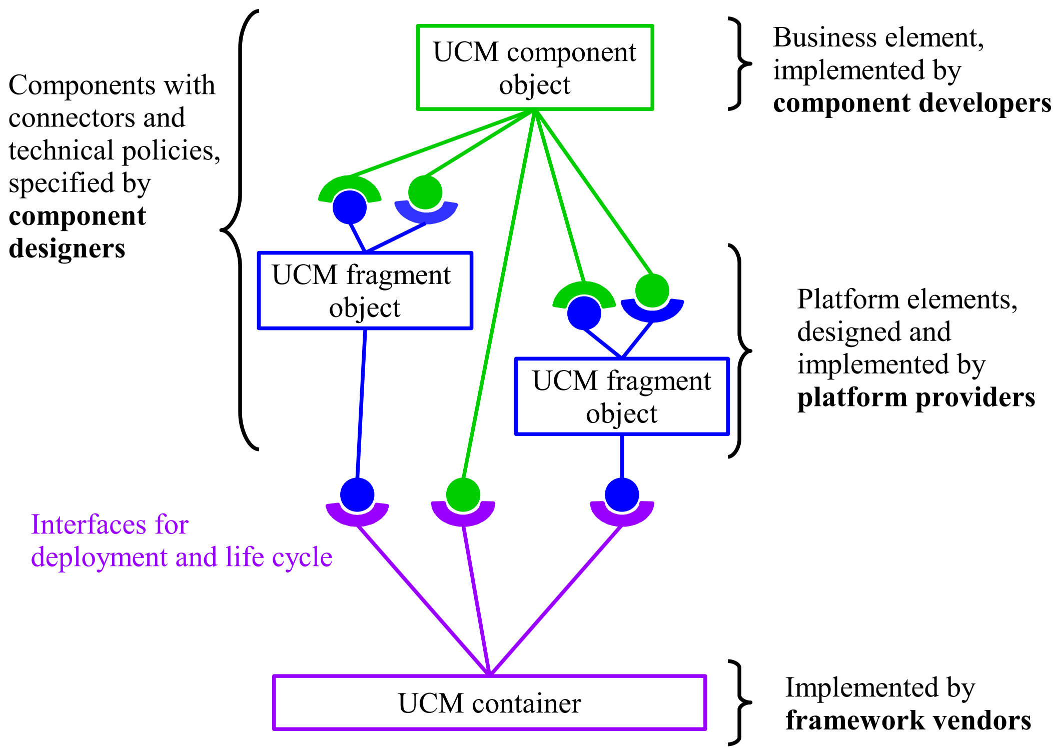

Figure 2: Component implementation architecture and its integration within a UCM infrastructure

1.5 UCM levels of conformance

UCM addresses several needs. The first is code portability, which implies API compatibility across frameworks and preservation of execution semantics with respect to real-time concerns. The second is extensibility to support domain-specific features (specific interaction mechanisms, runtime capabilities, etc.).

Minimal portability is ensured by the definition of UCM core specifications (§ 10.1), which address the basic interaction and technical policy APIs. All UCM platforms shall support the UCM core specifications. Any component that conforms with the UCM core specifications can therefore be executed on any UCM platform (provided the implementation language is supported). Core specifications only guarantee code portability; they do not enforce precise execution semantics.

Besides the core UCM specifications, platforms are capable of supporting additional capabilities, defined using UCM interaction and technical policy packages (§ 9.3and 9.4). Such platforms are then said to conform to the extensions of UCM.

2. Scope

For more than a decade, component-based software engineering has been considered a key enabler to increase software reuse and reduce time to market. The OMG developed the CORBA Component Model [CCM] as an enterprise component model for CORBA systems. It has been extended by a series of specifications to adapt it to different domains and provide additional capabilities ([QoS4CCM], [DDS4CCM], [AMI4CCM], [D&C]).

The Lightweight CCM (LwCCM) profile is one such extension, targeting embedded systems. A prime concern with the use of the LwCCM in embedded applications is that its mandatory dependency on the CORBA technology can lead to an undesirable memory and storage footprint, particularly when alternative middleware implementations are used.

These problems have led the Robotics Domain Task Force at the OMG to define its own standard to resolve some of these concerns, the Robotics Technology Component standard. Similarly, some CCM implementations have defined their own custom language mappings to circumvent the concern of the C++ language mapping.

This specification defines a Unified Component Model (UCM) as new component model targeting Distributed, Real-Time and Embedded (DRTE) Systems. UCM aims to be a simple yet complete, lightweight, middleware-agnostic, and flexible component model.

This specification defines a Platform Independent Model for UCM including:

- The definition of primitive and composite data types taking into account the main constraints encountered in DRTE developments and the need to master memory size on targets
- The definition of a functional component level allowing the design of software component architectures based on functional definitions of components and interaction patterns without any dependencies with the underlying technical environment.
- The definition of a Generic Interaction Support (GIS) based on connector principles allowing the specification of standard interaction patterns or the definition of specific patterns using generic mechanisms. This part is based on the GIS defined in the [DDS4CCM] specification.
- The definition of a component implementation level bringing hierarchical composition capabilities and allowing the refinement of functional components to fine grained segments supporting their own execution behavior.

The document also defines a standard programming model for business components and platform elements that must be implemented by PSMs. It specifies the generic mapping rules that apply to all classes that are part of the UCM PIM and specifically defines mappings to IDL and C++.

3. Rationale for a Unified Component Model

Several companies have adopted component-based software engineering for embedded real time or critical system. It has shown various benefits in terms of productivity and reusability, as it allows the definition of well-structured architectures and the use of code generation techniques. Due to domain constraints and the sometime very specific execution environments, companies often tend to build their own component model and associated frameworks, or make significant adaptations of existing standards (like lwCCM) to support these constraints.

This trend is due mainly to the non-functional aspects in DRTE (i.e. real time behavior, threading policies, memory allocation policies…) having a strong impact on the system behavior and can be very different from one domain to another. The capability to formally characterize these non-functional elements is mandatory to master behavioral analysis on the software architectures (WCET calculation, RT scheduling, data protection…) Moreover, the existing component models are usually defined with a specific underlying middleware and associated execution semantics that do not fit all DRTE environments.

These issues have led to the proposition of the Unified Component Model. UCM relies on a clear separation of architecture aspects between the specifications of platform capabilities and the design of application logic. It especially supports the capture of nonfunctional parameters, for generic or domain-specific concerns.

3.1 Separation of architecture concerns

The UCM approach to the design of software architectures consists of two parts, the definition of the platform capabilities (interactions and policies), and the specification of the functional elements (components), which will be controlled by the platform. These two parts are specified using concepts defined in the UCM meta-model (section 9).

3.1.1 Platform capabilities as model libraries

Platform capabilities are defined in model libraries, to be shipped with UCM tool chains. Connectors correspond to the communication capabilities provided by a UCM platform. They define the interaction logic between functional components. Technical policies correspond to the execution capabilities supported by a UCM platform. They define the technical aspects that can be associated with functional components (threading policy, clock, logging service, etc.).

Connector and technical policy definitions can have configuration parameters to specify nonfunctional settings related to the runtime implementation (e.g. execution periods, priorities, network addresses, etc.). As nonfunctional elements, configuration parameters are manipulated by the platform, but not by the component business code.

A minimal set of definitions is specified by the UCM standard in order to ensure portability of UCM components across UCM platforms. They cover standard interactions and standard technical policies. The UCM standard defines the semantics and APIs of these capabilities, but leaves their actual implementation middleware-dependent. The UCM standard thus guarantees portability across UCM platforms for functional code that relies on the minimal standard capabilities. Core UCM specifications are described in section 10.

Additional definitions may be provided by UCM platforms to support additional capabilities specific to a given domain or a given platform. UCM can thus support domain specific platform capabilities.

Connector and technical policy models ship with UCM platforms. They provide the specification of what is implemented in the corresponding UCM platform.

3.1.2 Business logic as components

Components correspond to the business logic. Nonfunctional elements such as thread management should not be handled by user code inside component bodies. Consequently, in UCM applications, all the functional code should be nested in components, without any direct call to runtime libraries. Explicit system calls in user code should be considered as bad practice, and limited to “technical” components that are not portable. It is good practice to integrate runtime libraries into technical policies, allowing the functional code (in the component body) and nonfunctional code (in a technical fragment) to interact through explicitly defined APIs; this eases code portability and integration.

Components can have attributes. Attributes are functional parameters that can be read (and written to, if allowed) by the business code.

3.2 Typical UCM process

A complete UCM process involves five main actors: UCM framework provider, UCM platform provider, component designer, component developer and software architect.

Infrastructure vendors provide a UCM framework and the associated libraries to support the execution of the business software. The UCM **platform provider** defines and implements interaction libraries (connectors, section 9.3) and container libraries (technical policies, section 9.4) with their APIs, configuration parameters and semantics. The UCM **framework provider** ships a tool set that is able to host and execute UCM components, connectors and technical policies. A UCM infrastructure is considered to be the backbone of every UCM application. A UCM framework typically ships with a set of connectors and technical policies (at least the core ones defined in section 10.1, but possibly additional ones).It might also allow the insertion of third-party libraries. Consequently, the platform provider and the framework provider may be one or many separate entities.

Users rely on the UCM platform to design and implement their component-based software application. The **component designer** first defines UCM functional contracts (section 9.2), then components (section 9.5), relying on connector definitions and technical policies to specify how components interact with their environment. From the component definitions, the **component developer** writes the content of the components, typically source code. The code is based on the APIs corresponding to the component specifications: it implements the component functional features and fulfills the component contracts.

Finally, the **software architect** defines the architecture of a particular domain application. This consists of assembling components, connectors and technical policies, specifying allocations on execution resources and setting values of configuration parameters.

The UCM standard provides all of the necessary concepts to support the work of the UCM platform provider, the UCM framework provider, the component designer and the component developer. The software architect shall use additional means to specify the component assembly and resource allocations.

4. Conformance

All UCM frameworks must support the UCM PIM defined in section 9. They also must at least ship with implementations of the core UCM platform specifications described in section 10.1. Implementation code must conform to the standard programming model (section 11).

UCM frameworks can ship with additional platform capabilities or implement extensions to the standard PIM in order to support specific application domains. Implementations of technical fragments may be specific to a given UCM framework by relying on additional specific APIs.

5. References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

5.1 Normative references

Given the relationship of UCM with CCM and its derived standards, this specification should be considered in relation to all the CCM-related specifications, namely:

- CORBA v3.3 – formal/2012-11-12, formal/2012-11-14, formal/2012-11-16
- LwCCM (part of CORBA v3.3) – formal/2012-11-16
- DDS for CCM [DDS4CCM] – formal/2012-02-01
- Quality of Service for CCM [QOSCCM] – formal/2008-10-12
- Deployment and Configuration of Component-based Distributed Applications [D&C] – formal/2006-04-02
- UML Profile for CCM – formal/2005-07-06

As UCM proposals are required to provide publish subscribe interaction patterns, this specification should be considered in relation with the DDS specification:

- Data Distribution Service [DDS] – formal/2007-01-01

As UCM proposals are required to provide an IDL PSM, the following formal specification should be considered:

- Interface Definition Language (IDL) 4.0 (BNF IDL 3.5 + ET) – mars/2013-05-32.

As UCM proposals are required to have an initial focus on the IDL C++11 language mapping. The formal specification of this latter should be considered:

- IDL to C++11 [IDLCPP11] – formal/2015-08-01

As UCM targets DRTE systems, this specification should be considered in relation to the existing OMG component standard for Robotics, namely:

- Robotic Technology Component, v1.1 [RTC] – formal/12-09-02

As UCM targets to be a successor of existing OMG component technologies this specification should be considered in relation to the existing OMG standard for Software Radio Components, namely:

- PIM & PSM for Software Radio Components [SDRP] - formal/2007-03-01

5.2 Non normative references

- [ECOA] European Common Operating System Collaboration Programme Architecture Specification Issue 3 http://www.ecoa.technology/public_specifications.html

6. Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Components are the functional elements of an architecture; they represent business logic. A component definition consists of two parts: a **component type** defines the ports for interaction with other components; a **component implementation** references a component type and specifies the internal structure of the component. There are two kinds of component implementations: **composite component implementations** are boxes with no execution semantics, they contain subcomponents to structure applications; **atomic component implementations** actually contain business logic.

Components communicate one with another through **interactions**. Interaction are specified in two steps. An **interaction pattern** defines the roles involved in the interaction (e.g. a client, a server) and the associated cardinality. A c**onnector definition** references an interaction pattern and defines the port APIs corresponding to the roles; it can also contain **configuration parameters** to specify nonfunctional settings (e.g. queue size, communication protocol).

Atomic component implementations can be associated with **technical policies**. Technical policies are implemented by component containers. They are defined in two steps. A **technical aspect** represents an abstract concept (e.g. a component life cycle). A **technical policy definition** is an actual specification of a technical aspect; it can define APIs to interact with the component, and can also contain **configuration parameters** (e.g. execution period).

Atomic component implementations consist of two parts: the **functional code** and the **technical code**. The functional code is the business logic of the component; it is nested in the **component body**. The technical code controls the business logic of the component; it is contained in technical **fragments** corresponding to bodies of **connectors** and **technical policies**. Fragments and component body are controlled by the **container**.

7. Symbols

UCM: unified modeling language

DRTE: distributed real-time and embedded

XML: extensible markup language

XMI: XML metadata interchange

IDL: interface description language

8. Additional Information

8.1 Acknowledgments

The following companies submitted this specification:

- THALES
- PrismTech Group Ltd

The following companies supported this specification:

- CEA – Commissariat à l'énergie atomique et aux énergies alternatives (French commission for atomic energy and alternative energies)

9. Platform Independent Model for UCM

The Unified Component Model defines a set of concepts that are used to specify software architectures made of interconnected functional components. All these concepts are formalized in a MOF-compliant meta-model that shall be implemented in UCM tools. The UCM meta-model is specified in document mars/16-05-05.

This chapter is the documentation of the UCM meta-model. It details the different entities defined by the meta-model.

9.1 Overview

9.1.1 Elements of the component model

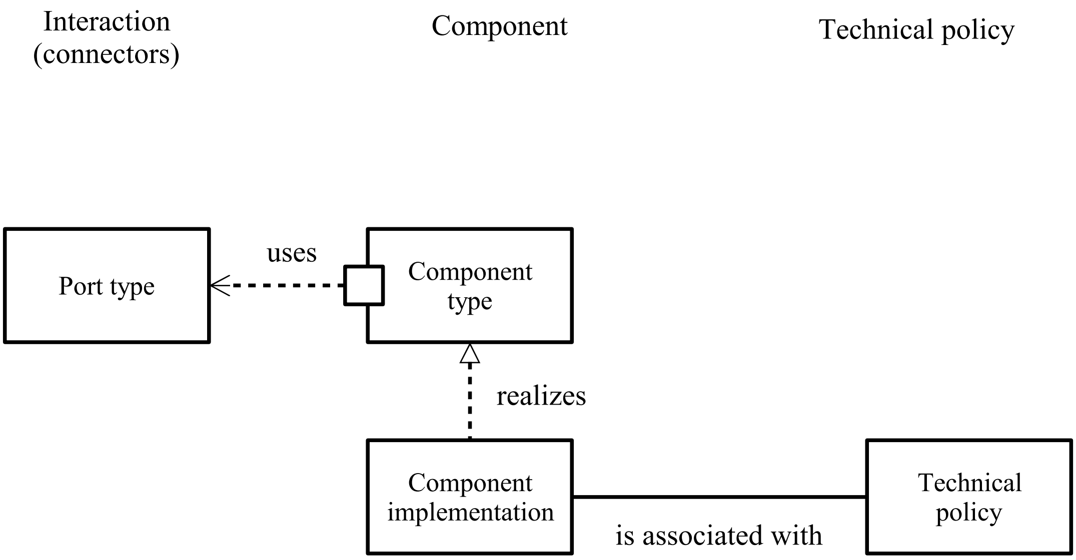

Figure 3: Relationship between components, connectors and technical policies

The Unified Component Model is decomposed into four main concerns:

- contracts and data types;
- components;
- connectors;
- technical policies.

Components encapsulate the application business logic. Connectors define the possible interactions between components. Technical policies define the possible interactions between the component business code and the underlying runtime libraries. Both connectors and technical policies define contracts that can be manipulated by the component business code. These contracts are attached to the components using ports (for connectors) or policies (for technical policies).

Components are the central entities in UCM. They contain business logic and rely on connector and technical policy definitions to specify interactions with their environment (see figure 3). Component types use ports types that are provided by connectors to interact with other components. Component implementations realize component types and are associated with technical policies to specify the possible interactions with the execution environment.

9.1.2 Configuration mechanisms

UCM provides three mechanisms to associate configurations with architecture entities: attributes, configuration parameters and properties. All three are specified in two steps: a definition and a value. They differ by their semantics and the entities they are defined in and associated with.

Attribute are functional elements: they can be manipulated by the business code nested in components. Configuration parameters are nonfunctional elements: they can be processed by framework tools, but are not seen by the business code. Properties are used to decorate functional elements; though they are not manipulated by the business code, they can be seen as formatted comments.

Attributes are defined in component definitions and interfaces. Their values are set in component instance configurations. For example, an interface that provides a method to compute the area of a circle from its diameter may

have an attribute to specify the value of π.

Configuration parameters are defined in declarations of platform entities: interaction patterns, connector definitions, connector implementations and technical policy definitions. Their values are set in the deployment plans—which are out of the UCM scope. For example, a technical policy that defines the periodic execution of a component may have a configuration parameter to specify the execution period.

Properties are defined in contract modules. They are associated with functional entities: methods, attributes, components (definitions and implementations) and component ports. For example, a component implementation may have a property to specify the revision number of its functional code.

9.1.3 Main packages of the meta-model

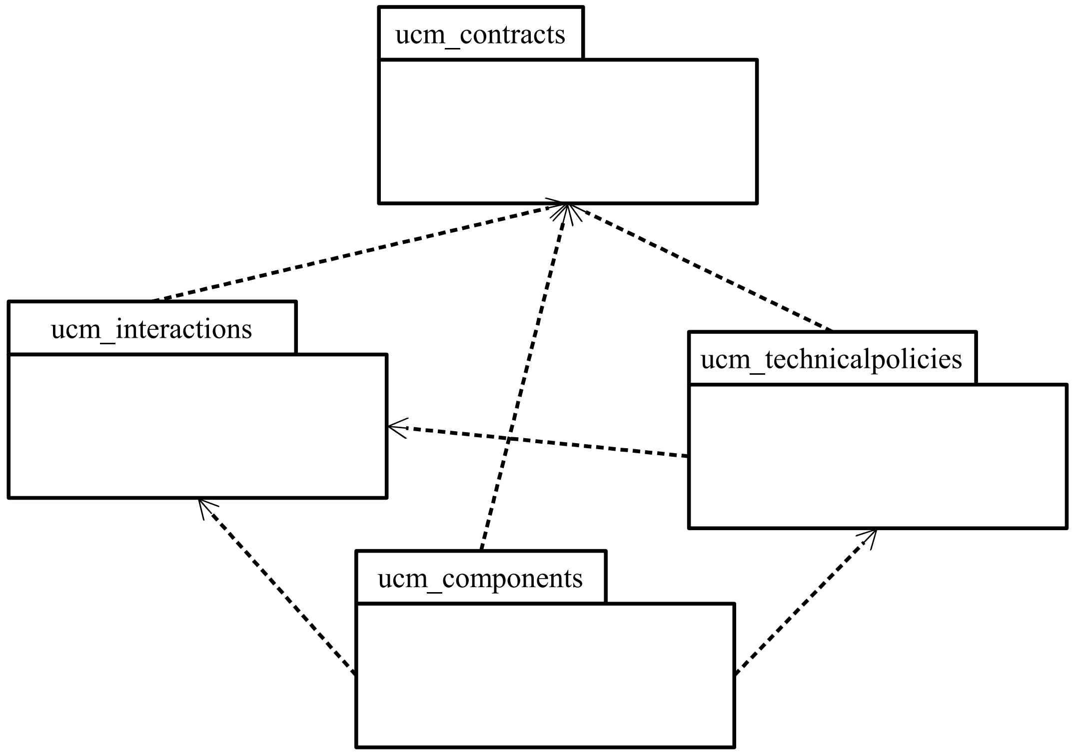

Figure 4: Main packages in the UCM meta-model

The meta-model is broken down into packages, each one focusing on a specific aspect. The main four packages are illustrated in 4 and are listed here, following a dependency order:

- ucm_contracts (§ 9.2)

  The contract package provides a description of how application entities can declare contracts for exchanging information. For instance, it supports the definition of data types and interfaces that provide an abstraction of the business domain. UCM defines a set of standard data types that are compatible with IDL data types.

- ucm_interactions (§ 9.3)

  The interactions package provides the necessary concepts for the definition of interaction patterns. An interaction pattern is a generic description of how application entities will interact, and how they can be connected through connectors realizing those patterns. This package depends on ucm_contracts to define contracts dedicated to local interaction between a component and the connector.

- ucm_technicalpolicies (§ 9.4)

  The technical policies package provides the necessary concepts for the definition of technical policies that represent requirements on component execution, and shall be ensured by the real-time architecture. Technical policies are typically implemented by containers. This package depends on ucm_contracts, as technical policies may have typed parameters or define APIs.

- ucm_components (§ 9.5)

  The components package defines the component model, which the description of application functional entities relies on. Those entities, called components, combine specifications of interaction patterns (from ucm_interactions) with contracts specifications (from ucm_contracts) to declare how they functionally interact.

This package depends on ucm_contracts to define application domain types that may be exchanged among components. It also depends on ucm_interactions and ucm_technicalpolicies, as it references interaction patterns and technical policies that apply to components

Packages ucm_interactions and ucm_technicalpolicies define the non-functional concepts implemented in UCM platforms (§ 10.1 for the standard definitions). Package ucm_components defines concepts used to define the functional part of architectures. All three package use the data types defines using the ucm_contracts package.

9.1.4 Common meta-model definitions

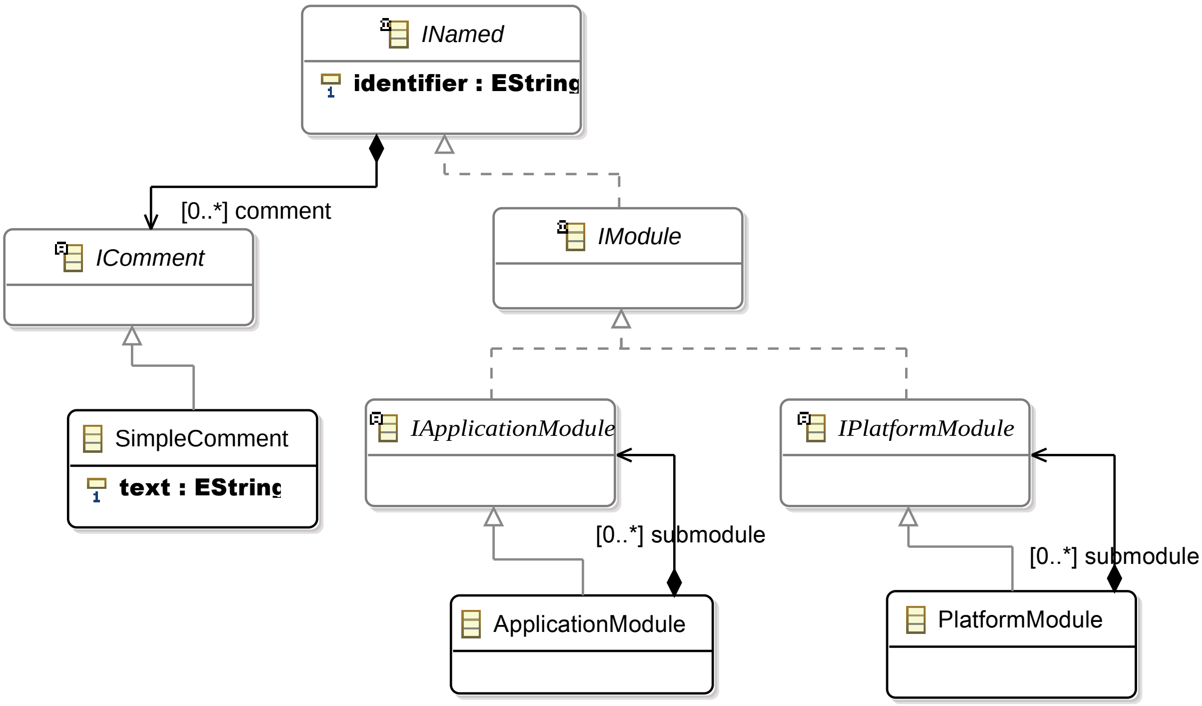

Figure 5: Base classes

A few classes are common ancestors for many others.

9.1.4.1 INamed

All classes that correspond to a named entity derive from abstract class INamed. Fields are:

- identifier: String [1…1]
- comment: IComment [0…*]

9.1.4.2 IComment

The purpose of abstract class IComment is to allow meta-model extensions for platform providers who would like to define alternative comment mechanisms.

9.1.4.3 SimpleComment (IComment)

Class SimpleComment is the only standard class to create comments. It consists of a string.

- Text: String [1…1]

9.1.4.4 IModule (INamed)

Abstract class IModule is the common ancestor for all module definitions. As it inherits INamed, all modules have a name and can contain comments.

A UCM model consists of a hierarchy of modules. There are two kinds of modules: application modules (component) and platform modules (interactions and technical policies).

9.1.4.5 IApplicationModule (IModule)

Abstract class IApplicationModule is the common ancestor of modules that contain application declarations: components and contracts.

9.1.4.6 IPlatformModule (IModule)

Abstract class IPlatformModule is the common ancestor of modules that contain platform declarations: interactions, technical policies and contracts.

9.1.4.7 ApplicationModule (IApplicationModule)

Class ApplicationModule can be used to gather several component and contract modules.

- submodule: IApplicationModule [0…*]

9.1.4.8 PlatformModule (IPlatformModule)

Class PlatformModule can be used to gather several interactions, technical policy and contract modules.

- submodule: IPlatformModule [0…*]

9.2 Contract package

9.2.1 Introduction

The contract package holds the definitions of contracts for UCM applications. Contracts mainly cover the definitions of interfaces and data types. The ucm_contracts package is complemented with a ucm_datatypes package that defines a meta-model for standard data types.

The contract package gathers several classes. A set of standard **data types** is defined; it is also possible to create meta-model extensions in order to define additional data types. **Constants** define specific values for a declared data type. **Interfaces** define consistent sets of methods related to a given service. The contract package also provides mechanisms to support the characterization of business and platform elements, using **annotations** and **configuration parameters**.

Among those declarations, only data types and interfaces are considered as types and can be used to specify interactions between components. Constants and exceptions are used to enrich the domain application specifications but do not directly contribute as the definition of contracts of interaction between components. Annotations and configuration parameters are used to decorate declarations.

9.2.2 Common definitions

The contract package contains a set of abstract classes that define the basic concepts carried by contracts: type declaration, annotation, configuration, etc. These abstract classes are extended by concrete classes; they can be used as hooks to support meta-model extensions.

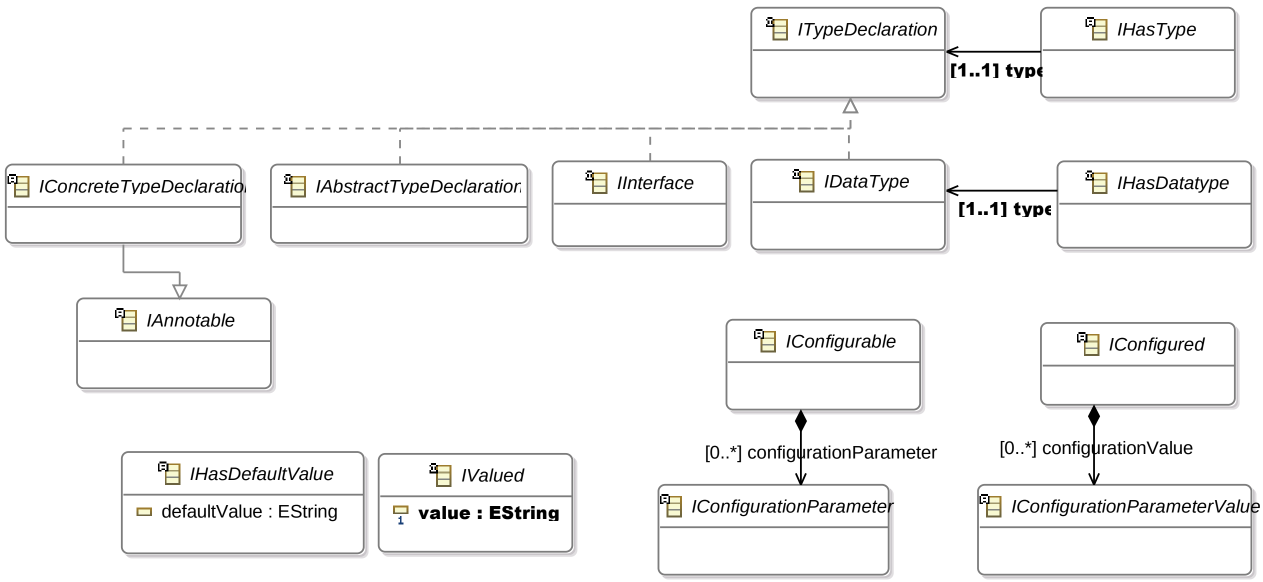

Figure 6: Abstract base classes

Figure 7 illustrates the definition of contract modules and the elements they contain. Contract modules mainly contain data type declarations and interface declarations. They also contain definitions of constants and exceptions, and annotations.

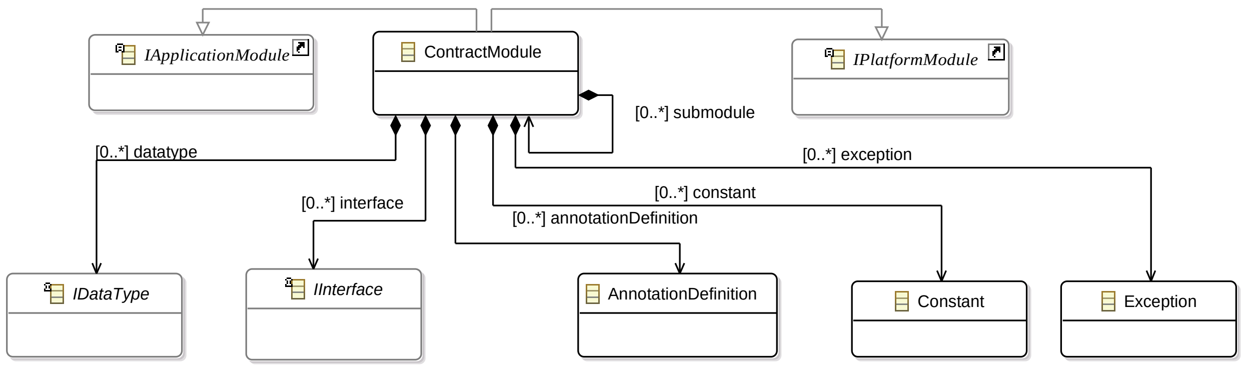

Figure 7: UCM contract base declarations

9.2.2.1 ITypeDeclaration (INamed)

Abstract class ITypeDeclaration is the common ancestor of all data type and interface declarations. As it inherits from Inamed, all UCM type declarations have a named: anonymous declarations are not possible in UCM.

9.2.2.2 IDataType (ITypeDeclaration)

Abstract class IDataTypeBase is the common ancestor of all data type declarations.

9.2.2.3 IInterface (ITypeDeclaration)

Abstract class IInterfaceBase is the common ancestor of all interface declarations.

9.2.2.4 IHasDataType

Abstract class IHasDataType is a base class used for entities that can reference data types (typically, composite data types or configuration parameters).

- type: IDataType [1…1]

Field type specifies the data type that is contained in the composite type. Abstract class IHasDatatype is used for concepts that must be typed with a data type – as opposed to with an Interface type. It is used for any type declaration that itself refers to another type declaration, as for instance, in array definition.

9.2.2.5 IHasType

Abstract class IHasType is a base class used for entities that can reference either data types or interfaces (typically, method parameters).

- type: ITypeDeclaration [1]

9.2.2.6 IValued

Abstract class IValued represents a concept that can accept a value. This class is the common ancestor for data type declarations that can have a value.

- value: string [1]

Field value is a plain string. IDL syntax must be used to specify values. See mars/2016-02-07.

9.2.2.7 IHasDefaultValue

Abstract class IHasDefaultValue is similar to abstract class IValued. It is used for default values, while IValued is used for actual values.

- defaultValue: string [1]

9.2.2.8 IAnnotable

Abstract class IAnnotable is the common ancestor for all classes that can have annotations. See section 9.2.9

- annotation: Annotation [0…*]

9.2.2.9 IAbstractTypeDeclaration

Abstract class IAbstractTypeDeclaration is the common ancestor for all classes that correspond to abstract types. See section 9.2.8.

9.2.2.10 IConcreteTypeDeclaration (IAnnotable)

Abstract class IConcreteTypeDeclaration is the common ancestor for all types that have actual semantics, as opposed to abstract types.

9.2.2.11 IConfigurationParameter (INamed)

Abstract class IConfigurationParameter is the ancestor of class ConfigurationParameter (§ 9.2.9). Its purpose is to allow meta-model extensions.

9.2.2.12 IConfigurable

Abstract class IConfigurable is the common ancestor of all classes that can define configuration parameters. See sections 9.3 and 9.4).

- configurationParameter: IConfigurationParameter [0…*]

9.2.2.13 IConfigurationParameterValue

Abstract class IConfigurationParameterValue is the ancestor of class ConfigurationParameterValue (§ 9.2.9). Its purpose is to allow meta-model extensions.

9.2.2.14 IConfigured

Abstract class IConfigured is the common ancestor of all classes that can specify configuration parameter values. See section 9.4).

- configurationValue: IConfigurationParameterValue [0…*]

9.2.2.15 ContractModule (IApplicationModule, IPlatformModule)

A contract module contains all kinds of declarations related with contracts: data types, constants, interfaces, exceptions. Contract modules can be nested in other contract modules to create hierarchies. Contract module also contain annotation definitions (§ 9.2.9). Fields are:

- submodule: ContractModule [0…*]
- datatype: IDataType [0…*]
- constant: Constant [0…*]
- exception: Exception [0…*]
- interface: IInterface [0…*]
- annotationDefinition: AnnotationDefinition [0…*]

Contract modules can be compared with IDL modules (building blocks Core Data Types and Basic Interfaces). They are used both for platform contracts and application contracts.

9.2.3 Standard data types: primitive data types

The UCM standard defines a set of primitive data types. Primitive types correspond to usual primitive data types of programming languages. These are integers, floating-point numbers, characters and Boolean. The semantics of UCM primitive data types are aligned with the definitions of IDL 4 Core Data Types building block.

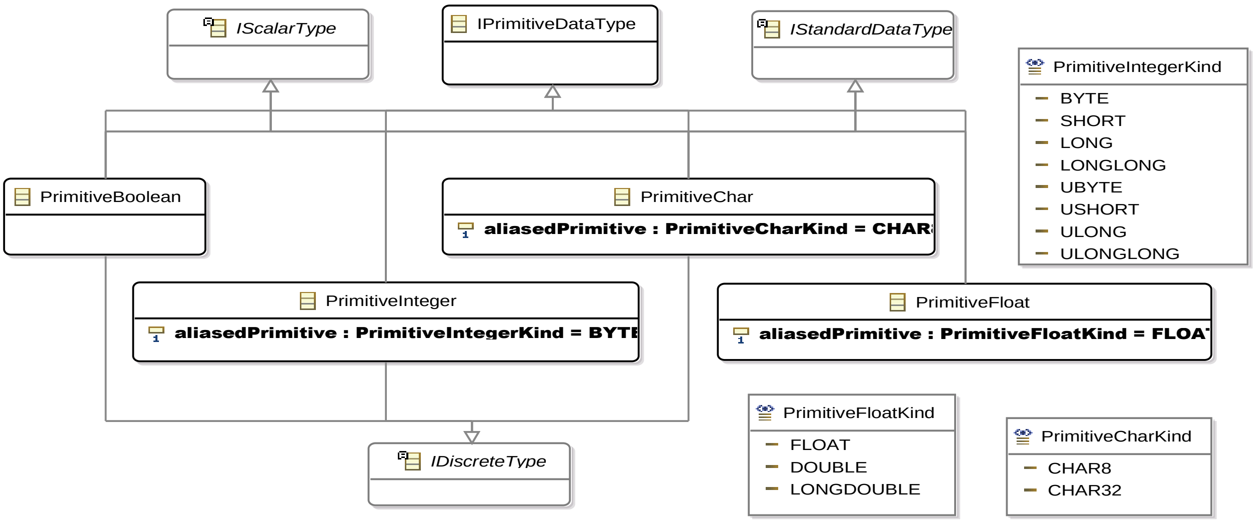

Figure 8: UCM primitive data types

9.2.3.1 IStandardDataType (IConcreteTypeDeclaration, IDataType)

Abstract class IStandardDataType is the common ancestor of all the UCM data types.

9.2.3.2 PrimitiveInteger (IStandardDataType, IPrimitiveDataType, IDiscreteType, IScalarType)

Class PrimitiveInteger corresponds to all kinds of integer types.

- aliasedPrimitive: PrimitiveIntegerKind [1]

Enumeration PrimitiveIntegerKind has these values: BYTE, SHORT, LONG, LONGLONG, UBYTE, USHORT, ULONG, ULONGLONG.

UCM integer type ranges are detailed in the following table:

UCM integer type	Lower bound	Upper bound	IDL equivalent
BYTE	-2^{7}	$2^{7}-1$	octet
SHORT	-2^{15}	$2^{15}-1$	short
LONG	-2^{31}	$2^{31}-1$	long
LONGLONG	-2^{63}	$2^{63}-1$	long long
UBYTE	0	$2^{8}-1$	char
USHORT	0	$2^{16}-1$	unsigned short
ULONG	0	$2^{32}-1$	unsigned long
ULONGLONG	0	$2^{64}-1$	unsigned long long

9.2.3.3 PrimitiveFloat (IStandardDataType, IPrimitiveDataType, IScalarType)

Class PrimitiveFloat corresponds to all kinds of floating-point types.

- aliasedPrimitive: PrimitiveFloatKind [1]

Enumeration PrimitiveFloatKind has the following values: FLOAT, DOUBLE, LONGDOUBLE.

The float types represent IEEE single-precision floating point numbers; the double type represents IEEE double-precision floating point numbers. For a detailed specification, see *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*.

There is no support for fixed-point values as there is first-class support for them in few languages and if present, it is often compiler-dependent (i.e., not part of the standard definition for the language).

9.2.3.4 PrimitiveChar (IStandardDataType, IPrimitiveDataType, IDiscreteType, IScalarType)

Class PrimitiveChar corresponds to all kinds of character types.

- aliasedPrimitive: PrimitiveCharKind [1]

Enumeration PrimitiveCharKind has two values: CHAR8 and CHAR32.

CHAR8 corresponds to an ASCII 8 bit encoded character. CHAR32 corresponds to a Unicode UTF-32 32 bit encoded character. No endianness convention is specified for CHAR32. The rationale for relying on UTF-32 rather than on UTF-8 or UTF-16 is that all UTF-32 characters have the same size, which eases the calculation of string length (even if UTF-32 is very expensive in terms of memory).

9.2.3.5 PrimitiveBoolean (IStandardDataType, IDiscreteType, IPrimitiveDataType, IScalarType)

Class PrimitiveBoolean corresponds to the boolean type.

9.2.4 Standard data types: complex types

Complex data types are aliases, arrays, structures, unions and enumerations.

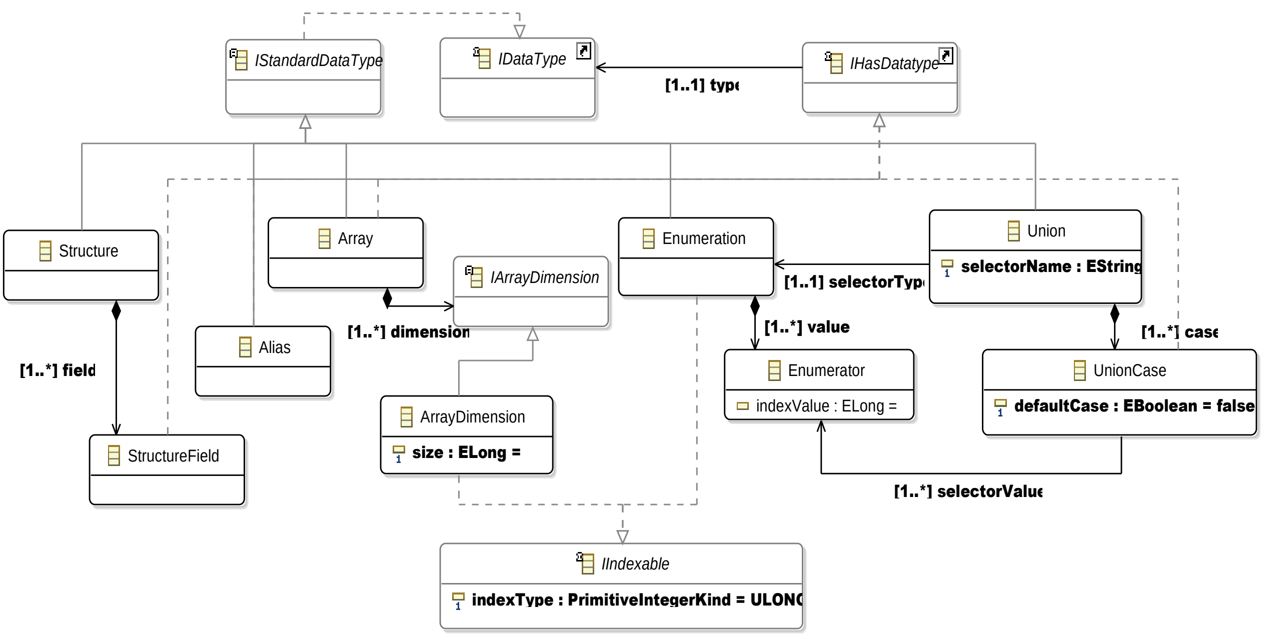

Figure 9: UCM complex data types

9.2.4.1 IIndexable

Abstract class IIndexable is the common ancestor for data types that contain several elements of the same type: sequences, strings, etc.

- indexType: PrimitiveIntegerKind [1]

Indexable types are indexed by an integer.

9.2.4.2 Alias (IStandardDataType, IHasDataType)

An alias type references another data type declaration. It is a way to rename data types.

9.2.4.3 Structure (IStandardDataType)

A structure declaration allows grouping heterogeneous types in fields. It has at least one structure field. Each field must have an identifier and a type.

- field: StructureField [1…*]

9.2.4.4 StructureField (INamed, IAnnotable, IHasDataType)

A structure field has a name and references a data type declaration.

9.2.4.5 Union (IStandardDataType)

A union is a data type that can take values from different data types. It has at least one union case. Each case represents the alternative fields for the value. To discriminate, at run time, which case is active, the union declares a selector (or discriminant) by specifying a selector name and a selector type.

- selectorName: String [1]
- selectorType: Enumeration [1]
- case: UnionCase [1…*]

The discriminant of a standard UCM union type is an enumeration. This is a limitation compared with some programming languages like Ada (which allow the use of any discrete type as discriminant); it ensure UCM union types can be mapped on any programming language.

9.2.4.6 UnionCase (INamed, IAnnotable, IHasDataType)

Class UnionCase contains a name and a data type. It also specifies the value of the selector for which it represents the union.

- selectorValue: Enumerator [1…*]
- defaultCase: boolean [1]

Cases must specify for which values of the selector they are active by setting the selector value. As unions are discriminated by an enumerated type, the selector values must be enumerators among the corresponding enumeration.

If field defaultCase is set to true, then the union case is used for all enumerators that are not used by other union cases. At most one union case per union type can be default.

9.2.4.7 Enumeration (IStandardDataType, IDiscreteType, IScalarType, IIndexable)

An enumeration is a type the values of which are known and finite in number. An enumeration is indexed, which means it must refer to an integer type from which it can take its values. An enumeration declares at least one enumerator that describes the accepted values for the enumeration.

- value: Enumerator [1…*]

9.2.4.8 Enumerator (INamed)

An enumerator corresponds to a value literal.

- indexValue: long [1]

The index value must be in the range of the primitive integer kind used as the index base for the enumeration.

9.2.4.9 Array (IStandardDataType, IHasDataType)

Array declarations represent a vector of entities of the same type, which size is fixed. Arrays can be multidimensionnal, each dimension having potentially different index types.

Arrays can have several dimensions.

- dimension: IArrayDimension [1…*]

9.2.4.10 IArrayDimension

Abstract class IArrayDimension is meant to allow meta-model extensions. For example, the UCM meta-model could be extended to allow array dimensions that specify a lower bound and an upper bound, or to allow array dimensions indexed by an enumeration.

9.2.4.11 ArrayDimension (IIndexable, IArrayDimension)

Class ArrayDimension specifies the dimension of an array.

- size: long [1]

Size is a long integer. As ArrayDimension inherits from IIndexable, size must be in the range of the underlying primitive integer. The corresponding array index ranges from 0 to size – 1

9.2.5 Standard data types: resizable types

A resizable data type is a data type the size of which can be adjusted.

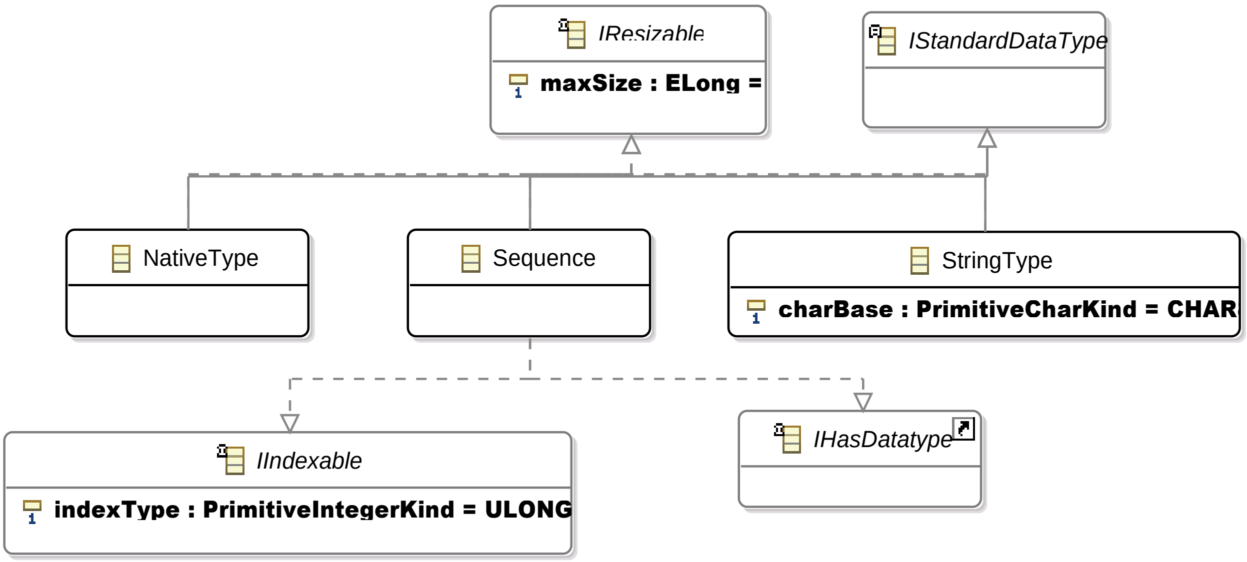

Figure 10: UCM resizable data types

9.2.5.1 IResizable

Abstract class IResizable is used for types that behave as collections of objects the size of which can vary. In order to respect the constraint that memory bound can be computed, this trait holds a property to define the maximum size. Even though the trait is called resizable, this doesn't entail any strategy for memory allocation and implementations can choose to use either dynamic allocation or to pre-allocate the maximum size buffer.

Class IResizable is the common ancestor of data types that have variable size, such as sequences.

- maxSize: long [1]

Where maxSize is a long integer. If there is no maximum size for the type, then maxSize should be set to "-1".

9.2.5.2 StringType (IStandardDataType, IResizable)

A string type is a string of characters, either 8-bit characters or 32-bit characters. Strings have a maximum bound; this bound can be set to "-1" for unbounded strings.

- charBase: PrimitiveCharKind [1]

9.2.5.3 NativeType (IStandardDataType, IResizable)

A native type represents a data type declaration specified using native constructions of a programming language. It has a maximum size, so that memory footprint can be computed without knowing the exact definition of the data type.

Field maxSize corresponds to the size of the underlying native type, in bytes.

A native type represents a data type that is not represented in the UCM model but that is to be used within UCM applications. Native types have several use cases, the main two being:

- Representing types available in a language that can't be represented with UCM type model;
- Representing types that are used at the frontier of integration of a UCM-based application and an external one.

Whatever useful, it is recommend to avoid the use of native types, as they lead to major portability issues.

9.2.5.4 Sequence (IStandardTypeBase, IHasDataType, IResizable, IIndexable)

Sequence declarations represent a vector of entities of the same type, the size of which can vary between 0 and maxSize.

Sequences can be seen as one-dimension arrays with a variable size. Their size is bounded. If its field maxSize is set to "-1", the sequence is unbounded.

9.2.6 Constants

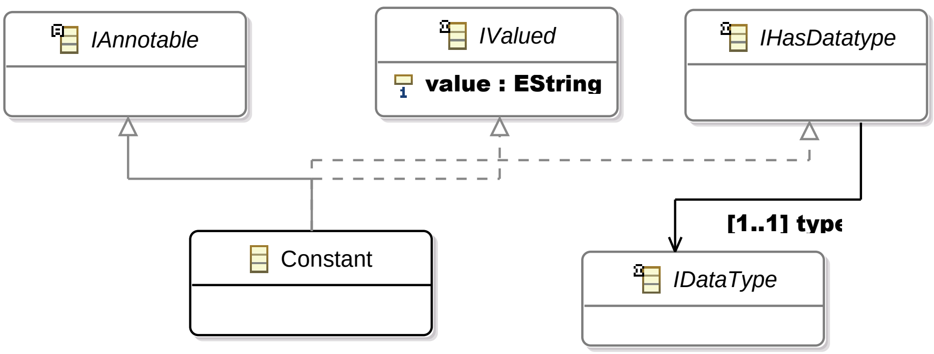

Figure 11: UCM constants

9.2.6.1 Constant (INamed, IHasDataType, IValued, IAnnotable)

Constant declaration only requires an identifier, a type and a value.

The data type value is a string that follows the IDL grammar dedicated to specifying values of constants. See building block Core Data Type in mars/2016-02-07.

9.2.7 Interfaces, methods and exceptions

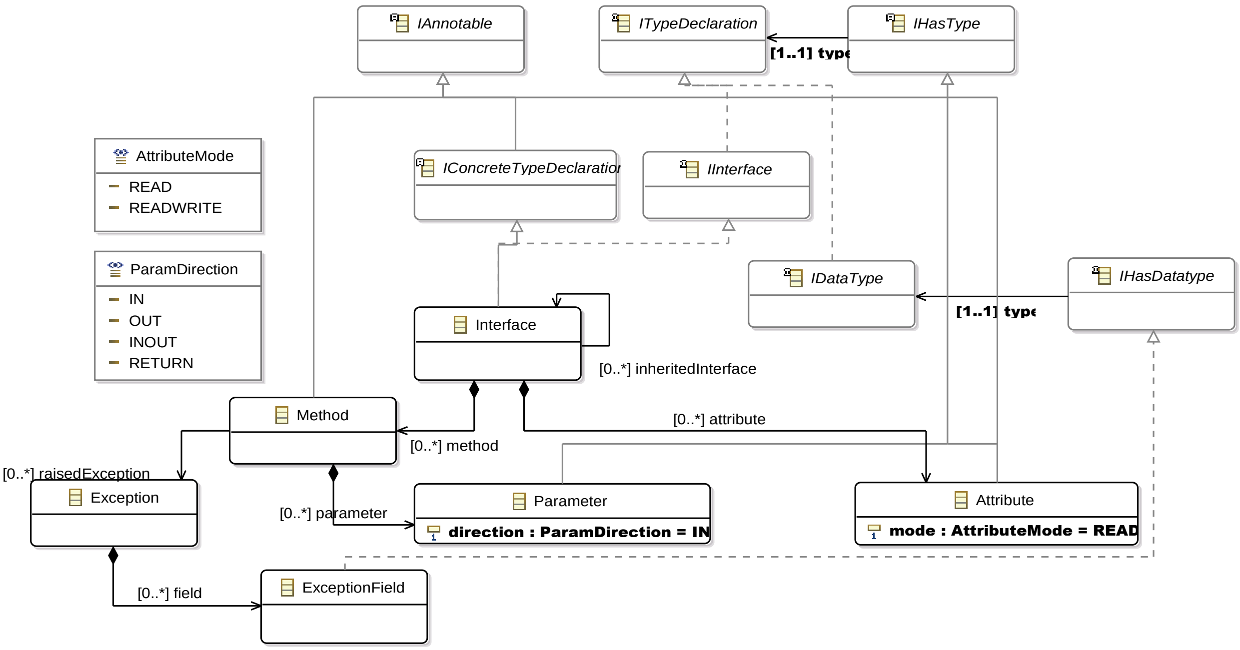

Figure 12: UCM interfaces

An **interface** allows for declaring a consistent set of functions related to a given service. An interface has 0 or more **attributes** that hold the state of the interface instance. These attributes have a **mode** that specifies the read/write access. An attribute may also have a default value; the syntax for this default value must follow the grammar used for Constants – see section 9.2.6.

An interface has 0 or more **methods** that define actions possible on that object. A method only declares a signature, as a list of **parameters** that have a **direction** among: **IN**, **OUT**, **INOUT** and **RETURN**. Methods have at most one parameter with direction **RETURN**. Methods can also have **exceptions**, which correspond to return codes in case of abnormal execution.

Interfaces refer to zero or more interfaces called **inherited interfaces**.

An **exception** declaration defines a kind of structure holding error information. This declaration is only used inside interface declaration (see next sub-section) to specify how an interface method can fail and which failure details it should provide to the caller. For that purpose, an exception has zero or many **exception fields** that have an identifier and refer to a data type.

The notion of exception in UCM must not be confused with the notion of exception in programming languages. Indeed, UCM exceptions are only data structures that must be provided to callers in case of abnormal execution. No assumption is made regarding the way such data structures are transmitted to callers: this might be through plain exception mechanism or through extra output parameters. The solution to choose is mapping-dependent.

9.2.7.1 Interface (IinterfaceBase, IConcreteTypeDeclaration)

- inheritedInterface: Interface [0…*]
- attribute: Attribute [0…*]
- method: Method [0…*]

An interface can inherit other interfaces. In these situations, the interface contains its own methods and attributes, plus the methods and attributes of its ancestors.

Attributes are shortcuts to define access methods (get and set). They do not necessarily correspond to actual data.

9.2.7.2 Method (INamed, IAnnotable)

- parameter: Parameter [0…*]
- raisedException: Exception [0…*]

9.2.7.3 Parameter (INamed, IHasType)

- direction: ParamDirection [1]

Enumeration ParamDirection contains the following values: in, out, inout, return.

A parameter that has direction "return" is a return type of the method. Consequently, A given method can have at most one "return" parameter.

9.2.7.4 Attribute (INamed, IHasType, IAnnotable, IHasDefaultValue)

- mode: AttributeMode [1]

AttributeMode is an enumeration that can have the following values: read, readwrite.

9.2.7.5 Exception (INamed)

- field: ExceptionField [0…*]

9.2.7.6 ExceptionField (INamed, IHasDataType)

An exception field is similar to a structure field.

9.2.8 Abstract type declarations

Besides explicit data type and interface declarations, the UCM data model defines two additional declarations: AbstractDataType and AbstractInterface. They are to be used as replacement for actual type declarations in port types (§ 9.3.5) and technical policy definitions (§ 9.4.3); they are eventually bound to an actual data type or interface (§ 9.5.3.7).

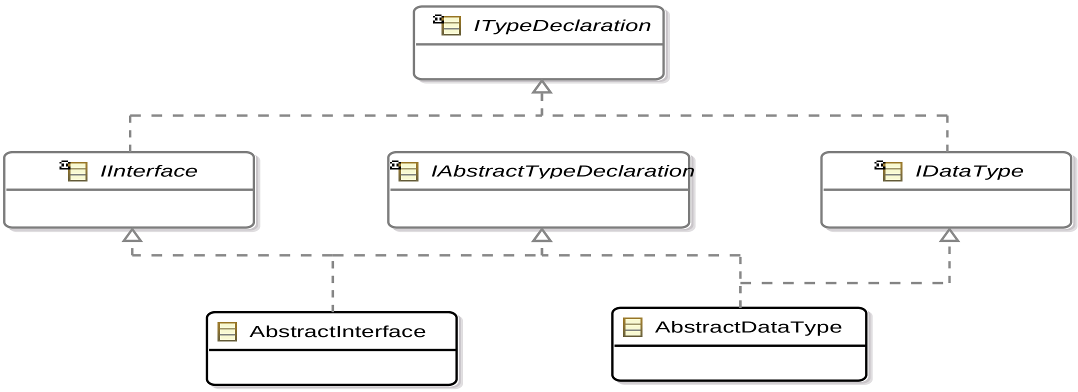

Figure 13: UCM template parameters

9.2.8.1 AbstractDataType (IAbstractTypeDeclaration, IDataType)

Class AbstractDataType is used for the declaration of a generic data type.

9.2.8.2 AbstractInterface (IAbstractTypeDeclaration, IInterface)

Class AbstractInterface is used for the declaration of a generic interfaces.

9.2.9 Annotations and configuration elements

UCM supports two mechanisms to specify architecture configuration: configuration parameters and annotations. Configuration parameters apply to platform elements (connectors and technical policies) while annotations can be associated with business elements (components, interfaces, methods, etc.).

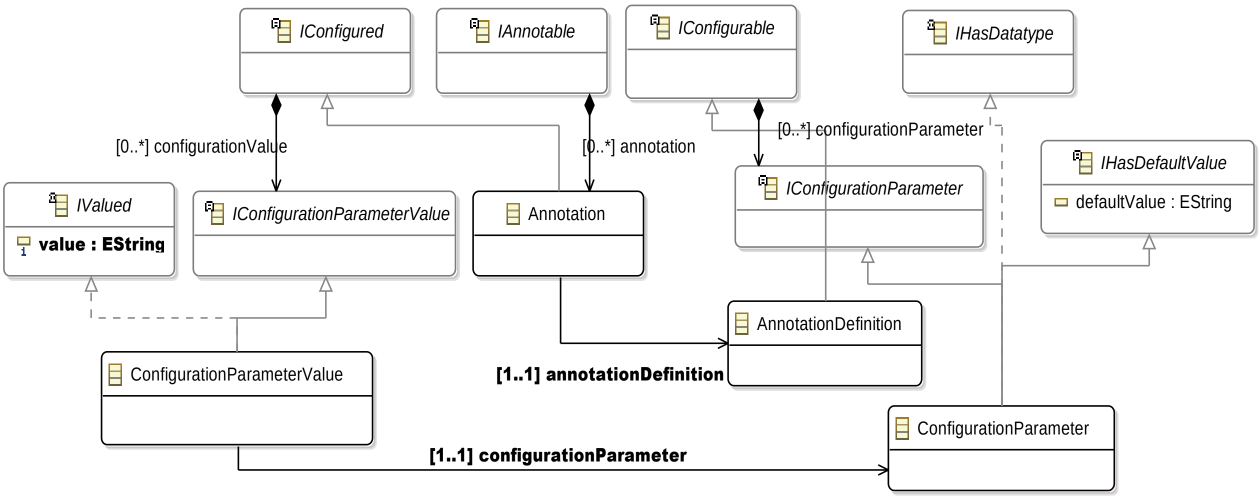

Figure 14: UCM configuration elements

9.2.9.1 ConfigurationParameter (IConfigurationParameter, IHasDataType, IHasDefaultValue)

Configuration parameters are comparable to attributes. Attributes are functional elements, and therefore can be manipulated by business code. Configuration parameters are nonfunctional elements: they have no direction, as they are properties associated with platform elements. They cannot be manipulated by code, but are typically used to create or configure the platform code.

Values of configuration parameters should be specified in deployment models, which is out of the scope of the UCM standard.

9.2.9.2 ConfigurationParameterValue (IValued, IConfigurationParameterValue)

A configuration parameter value associates a value to a configuration parameter definition.

- configurationParameter: ConfigurationParameter [1]

9.2.9.3 AnnotationDefinition (INamed, IConfigurable)

Class AnnotationDefinition contains a set of configuration parameters. Though annotations do not apply to platform elements, annotation definitions contain a set of configuration parameters. This is for metamodel factorization.

9.2.9.4 Annotation (IConfigured)

An Annotation references an annocation definition. It is used to set values to the parameters declared in the annotation definition.

- annotationDefinition: AnnotationDefinition [1]

An annotation can be seen as a formatted comment.

9.3 Interactions package

9.3.1 Overview

The UCM meta-model is independent from any specific communication middleware. Middleware specific declarations can be provided as predefined elements. To do so, UCM defines a Generic Interaction Support (GIS) inspired by the CCM GIS.

The UCM standard specifies a generic mechanism for the definition of interactions between components. The ucm_interactions package has three main goals:

- Specify **roles** and **items** involved in an **interaction pattern**.
- Specify **port types**, carried by **connectors**, to define explicit API.
- Specify **configuration parameters**, also carried by connectors, to support the configuration of the underlying middleware.

Interaction patterns define the overall logic of an interaction. They define a set of roles involved in the interaction (e.g. data producer, data consumer) and the number of entities that can have these roles in the interaction (e.g. a unique producer, one or more consumers).

Connector definitions are refinements of interaction patterns. They define ports that associate APIs to roles. A connector definition therefore defines the programming contracts involved in an interaction. A connector definition specifies the semantics and API for a given interaction pattern. Several connector definitions can reference the same interaction pattern.

Figure 15 provides a simple example. It illustrates the specification of an interaction pattern named “example_interaction_pattern”, which defines two roles: “emitter” and “receiver”. The pattern specifies that in an interaction of kind “example_interaction_pattern”, there is at least one emitter and at least one receiver. Both roles manipulate an item named “data_item”; this means this interaction transfers one piece of data. At this stage, no information is provided regarding APIs.

A connector definition named “example_connector” is associated with the interaction pattern. It specifies the API that business code shall use: emitters will require interface “api_itf” while receivers will provide the same interface. Item “data_item” is bounded to an abstract data type named “data_type_t”, which is referenced by the interface. The connector has a configuration parameter named “socket_config”, which is used to specify the kind of socket to use for communications: Unix socket or IP socket.

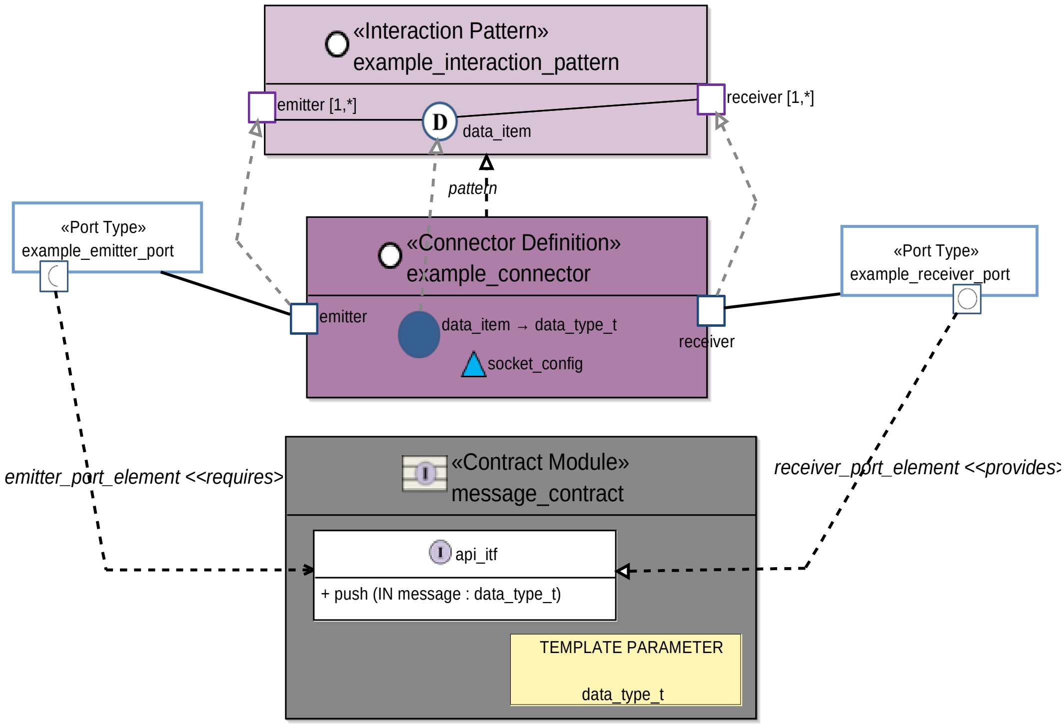

Figure 15: Example of interaction pattern and connector definition

The corresponding XML representation is the following:

```
<InteractionModule name="intr_mod">
  <ContractModule name="intr_contr_mod">
    <Enumeration indexType="BYTE" name="socket_kind">
      <Enumerator name="socket_inet" value="0"/>
      <Enumerator name="socket_unix" value="1"/>
    </Enumeration>
    <AbstractDataType name="data_type_t"/>
    <Interface name="api_itf">
      <Method name="push">
        <Parameter direction="IN" name="message" type="data_type_t"/>
      </Method>
    </Interface>
  </ContractModule>
  <InteractionPattern name="intr_pat_1">
    <InteractionRole lowerMultipliciy="1" name="emitter" upperMultipliciy="-1">
      <InvolvedItem item="data_item"/>
    </InteractionRole>
    <InteractionRole lowerMultipliciy="1" name="receiver" upperMultipliciy="-1">
      <InvolvedItem item="data_item"/>
    </InteractionRole>
    <InteractionItem name="data_item" nature="DATA"/>
  </InteractionPattern>
  <PortType name="emit_prt">
    <PortElement interface="intr_contr_mod::api_itf" kind="REQUIRED"
                 name="emitter_prt_elem"/>
  </PortType>
  <PortType name="receive_prt">
    <PortElement interface="intr_contr_mod::api_itf" kind="PROVIDED"
```

```
                 name="receive_prt_elem"/>
  </PortType>
  <ConnectorDefinition name="cnt_def_1" pattern="intr_pat_1">
    <ConfigurationParameter defaultValue="socket_inet" name="socket_config"
                            type="intr_contr_mod::socket_kind"/>
    <ConnectorPort api="emit_prt" name="emitter" portRole="intr_pat_1.emitter"/>
    <ConnectorPort api="receive_prt" name="receiver"
portRole="intr_pat_1.receiver"/>
    <ItemBinding connectorItem="intr_contr_mod::data_type_t"
                 interactionItem="intr_pat_1.data_item"/>
  </ConnectorDefinition>
</InteractionModule>
```

The IDL definition of the connector API is the following:

```
module intr_mod<type data_type_t> {
 module intr_contr_mod {
  interface api_itf {
   void push (in data_type_t message);
  }
  enum socket_kind {socket_inet, socket_unix};
 }
 port type emit_prt {
  requires message_contract::api_itf emitter_port_element;
 }
 port type receive_prt {
  provides message_contract::api_itf receiver_port_element;
 }
 connector example_connector {
  mirror port emit_prt emitter;
  mirror port receive_prt receiver;
 }
}
```

The main entities defined in the ucm_interaction package are illustrated in figure 16.

9.3.2 Interaction module

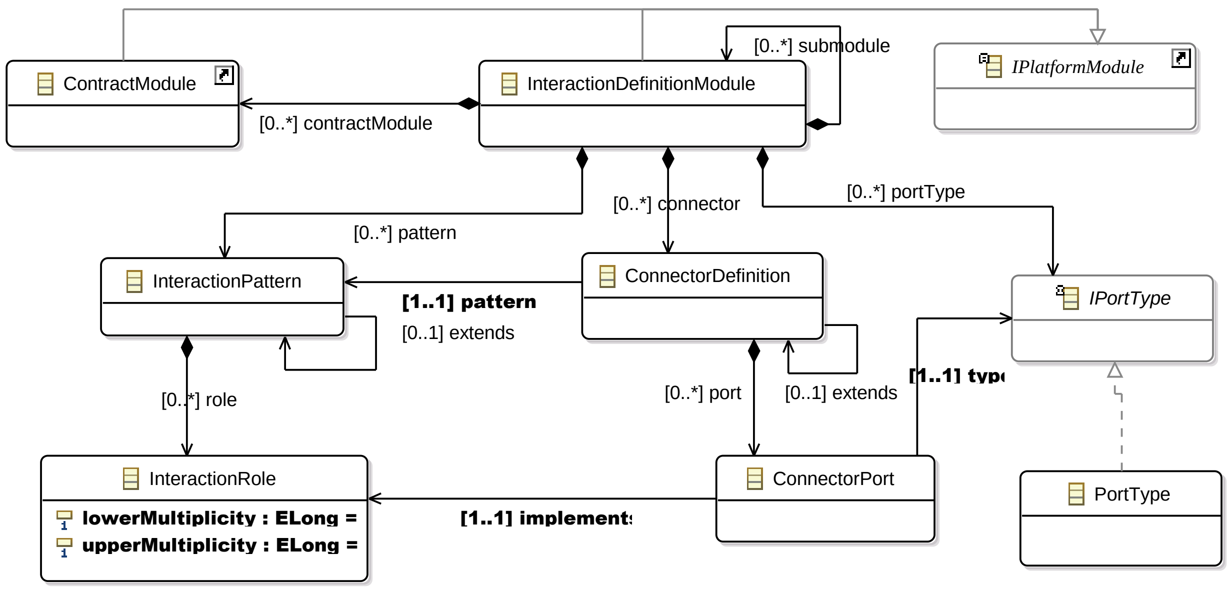

Figure 16: Main classes of the UCM interaction package

9.3.2.1 InteractionDefinitionModule (IPlatformModule)

Interaction definition modules contain the definitions of the possible interactions between components. In other words, they contain the specification of the UCM interaction logics from an application point of view that can be used in a given architecture. An interaction definition module has the following information:

- contractModule: ContractModule [0…*]
- submodule: InteractionDefinitionModule [0…*]
- pattern: InteractionPattern [0…*]
- connector: ConnectorDefinition [0…*]
- portType: IPortType [0…*]

Interaction definition modules can have submodules, to allow hierarchical definitions. They can also contain contract modules to store data types and interface definitions directly associated with the interaction definitions.

9.3.2.2 XML representation

The XML schema for interaction definition modules is the following:

```
<xsd:element name="InteractionModule" type="InteractionModuleType" />
<xsd:complexType name="InteractionModuleType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="ContractModule"
type="ContractModuleType" />
    <xsd:element maxOccurs="unbounded" name="InteractionPattern"
type="InteractionPatternType" />
    <xsd:element maxOccurs="unbounded" name="PortType" type="PortTypeType" />
    <xsd:element maxOccurs="unbounded" name="ConnectorDefinition"
type="ConnectorDefinitionType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="extends" type="xsd:string" />
</xsd:complexType>
```

9.3.2.3 IDL equivalent syntax

An interaction module is represented by an IDL **module**. Such an IDL module must only contain connectors and port types.

9.3.3 Interaction patterns

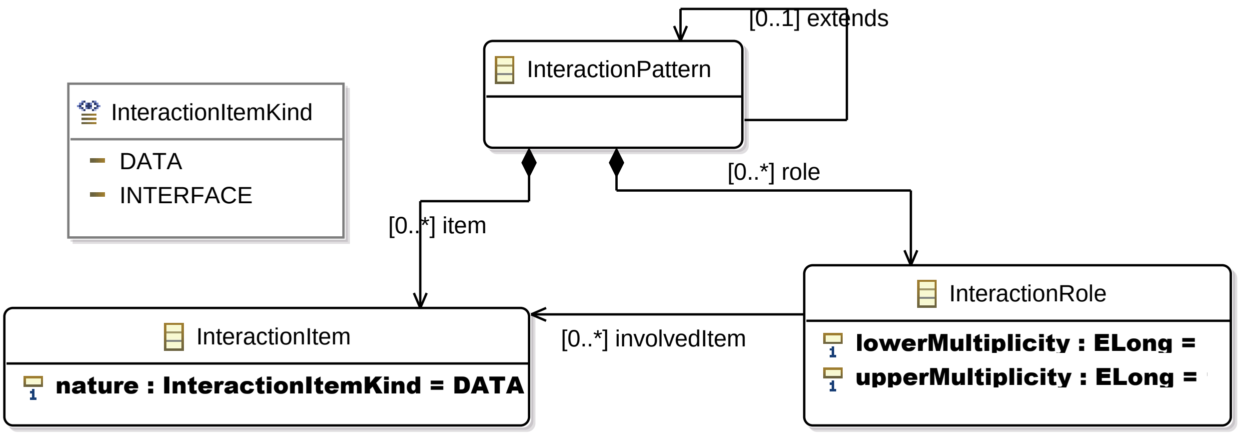

Figure 17: UCM interaction patterns

Interaction patterns provide a definition of the roles different participants can have in an interaction. These roles do not entail any API; they only provide high-level semantics on which one can rely to define assemblies of components.

Designing an interaction pattern involves the combination of different entities that play different roles. For instance, a publish / subscribe interaction pattern combines several publishers with several subscribers. A streaming interaction pattern combines one writer with several readers. This notion of role is thus the placeholder for:

- A multiplicity that tells how many entities can have a given role;
- An identifier that bears the semantic of that role;
- Interaction items related to this role.

9.3.3.1 IInteractionDefinition (INamed)

Abstract class IInteractionDefinition is used as a common ancestor for both InteractionPattern and ConnectorDefinition. This allows the specification of inter-component connections that can either reference a connector or an interaction pattern. See section 9.5.5.4.

9.3.3.2 InteractionPattern (IInteractionPattern)

An interaction pattern is the main declaration entity. It defines the relationship between roles. It also indicates elements that are manipulated by the interaction.

- role: InteractionRole [0…*]
- item: InteractionItem [0…*]
- extends: InteractionPattern [0…1]

  An interaction pattern can extend another interaction pattern to define additional roles. Roles cannot be redefined.

9.3.3.3 InteractionItem (INamed)

Interaction items are used to specify the items manipulated by an interaction pattern. They are used to specify flows through interaction patterns, to help ensure consistency when defining connectors.

- nature: InteractionItemKind [1]

InteractionItemKind is an enumerated type that has two possible values: “data” and “interface”. Hence, an interaction item defines a name that shall correspond either to a data type definition or to an interface definition.

9.3.3.4 InteractionRole (INamed)

- lowerMultiplicity: long [1]

- upperMultiplicity: long [1]
- invlovedItem: InteractionItem [0…*]

lowerMultiplicity and upperMultiplicity specify how many times the given role can be involved in a given interaction pattern. Field invlovedItem associates interaction items with the role. Roles that are associated with the same item shall correspond to connector ports that manipulate the same data type or interface.

9.3.3.5 Graphical representation

An interaction pattern is represented by a light purple box. Its symbol is a circle.

9.3.3.6 XML representation

```
 <xsd:complexType name="InteractionPatternType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="InteractionRole"
type="InteractionRoleType" />
      <xsd:element name="InteractionItem" type="InteractionItemType" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="InteractionItemType">
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="nature" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="InteractionRoleType">
    <xsd:sequence>
      <xsd:element name="InvolvedItem" type="InvolvedItemType" />
    </xsd:sequence>
    <xsd:attribute name="lowerMultipliciy" type="xsd:int" />
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="upperMultipliciy" type="xsd:int" />
  </xsd:complexType>
  <xsd:complexType name="InvolvedItemType">
    <xsd:attribute name="item" type="xsd:string" />
  </xsd:complexType>
```

9.3.3.7 IDL equivalent syntax

As interaction patterns define no API, there is no equivalent IDL syntax for them.

9.3.4 Connector definitions

Connectors refine interaction pattern to specify explicit APIs and middleware configuration parameters.

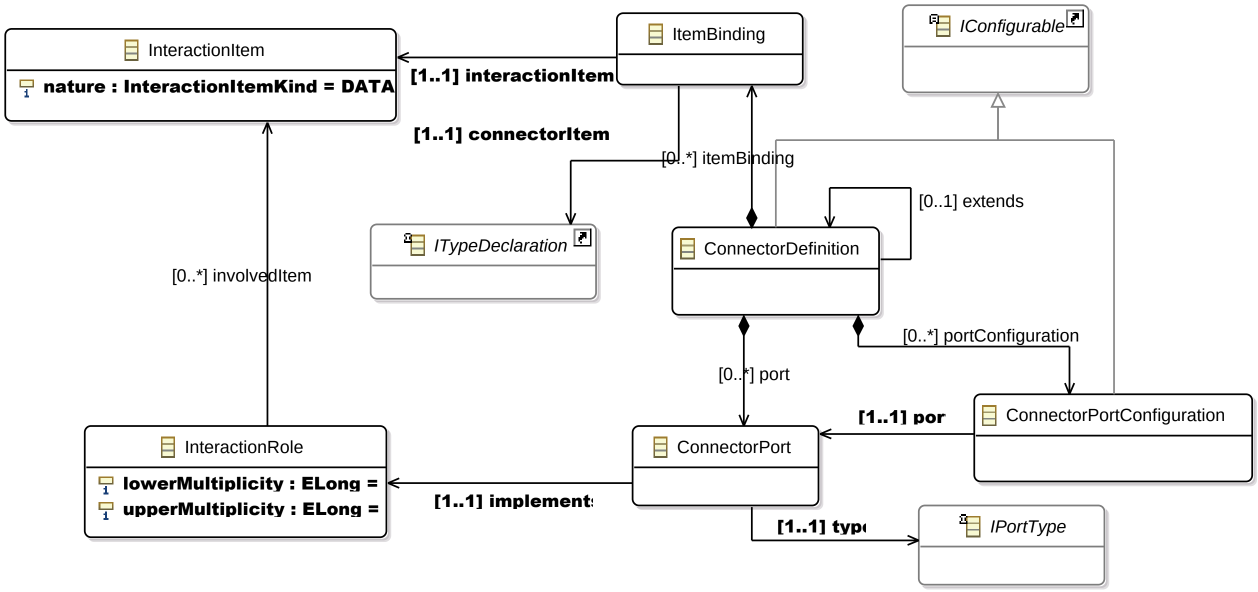

Figure 18: UCM connectors

9.3.4.1 ConnectorDefinition (IInteractionDefinition, IConfigurable)

Connector definitions specify possible interactions from a business point of view. That is, they describe the functional ports involved in a given interaction and the parameters of this interaction. A connection definition has the following information:

- pattern: InteractionPattern [1]
- port: ConnectorPort [0…*]
- itemBinding: ItemBinding [0…*]
- portConfiguration: ConnectorPortConfiguration [0…*]
- extends: ConnectorDefinition [0…1]

A connector can refine another connector definition to add ports or configuration parameters.

Configuration parameters allow for the specification of nonfunctional parameters of the whole connector (e.g. the specification of a channel name). Port configurations have the same purpose, but dedicated to a given port (e.g. the specification of a FIFO size).

9.3.4.2 ItemBinding

Connectors have to specify which data types or interfaces interaction items are bound to. This is a way to ensure consistency between the high level specifications of interaction patterns and detailed APIs of connectors: a connector must associate all the items of its interaction pattern to data types or interfaces manipulated in its ports. An ItemBinding has the following information:

- interactionItem: InteractionItem [1]
- connectorItem: ITypeDeclaration [1]

9.3.4.3 ConnectorPort (INamed)

Connector ports correspond to the interaction points of a connector. They define the interaction APIs that will be offered to components and used through component ports. A connection port definition has the following information:

- implements: InteractionRole [1]
- type: IPortType [1]

A connector port references an interaction role of the interaction pattern referenced by the connector. The connector port thus relies on the multiplicity defined for the corresponding role. This enables the definition of several ports for a given role without confusions.

9.3.4.4 IPortType (INamed)

This class is abstract and corresponds to the specifications of detailed port API. In the UCM standard, it is extended by class PortType. Extensions of the UCM standard could define other concrete classes to specify APIs.

9.3.4.5 Graphical representation

A connector definition is represented by a purple box. The associated symbol is a circle.

9.3.4.6 XML representation

```
 <xsd:complexType name="ConnectorDefinitionType">
    <xsd:sequence>
      <xsd:element name="ConfigurationParameter"
type="ConfigurationParameterType" />
      <xsd:element maxOccurs="unbounded" name="ConnectorPort"
type="ConnectorPortType" />
      <xsd:element name="ItemBinding" type="ItemBindingType" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="pattern" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="ItemBindingType">
    <xsd:attribute name="connectorItem" type="xsd:string" />
    <xsd:attribute name="interactionItem" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="ConnectorPortType">
    <xsd:attribute name="api" type="xsd:string" />
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="portRole" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="ConfigurationParameterType">
    <xsd:attribute name="defaultValue" type="xsd:string" />
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="type" type="xsd:string" />
  </xsd:complexType>
```

9.3.4.7 IDL equivalent syntax

A connector definition is represented by an IDL **connector**. A connector port is represented by an IDL **mirror port**.

9.3.5 Port definitions

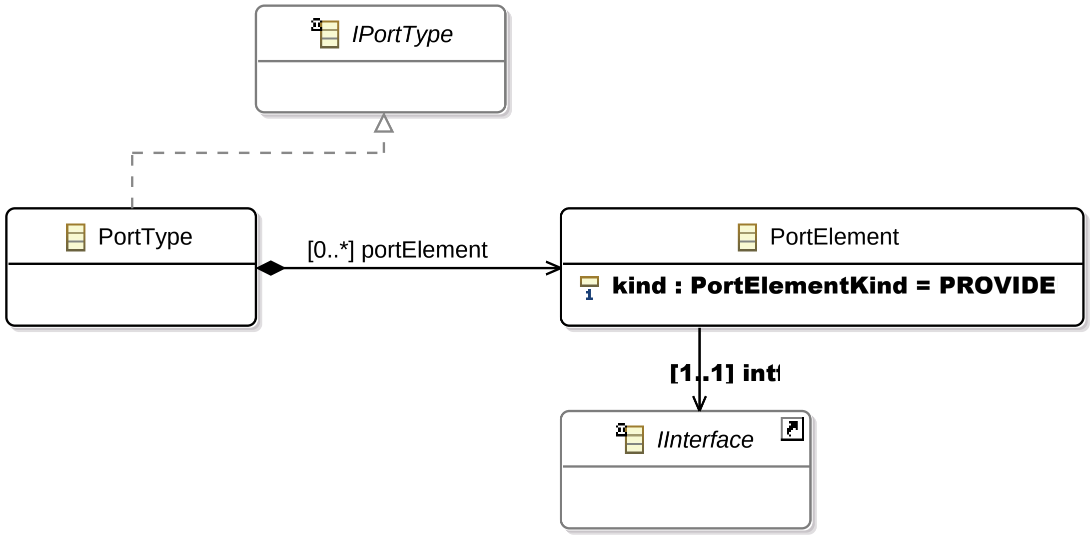

Figure 19: UCM port types

9.3.5.1 PortType (IPortType)

A port type is a concrete realization of the IPortType class. It defines a set of port elements.

- portElement: PortElement [0…*]

9.3.5.2 PortElement (INamed)

A port element either provides or require an interface.

- intf: IInterface [1]
- kind: PortElementKind [1]

It references an interface. PortElementKind is an enumerated type that has two values: “provided” or “required”.

9.3.5.3 Graphical representation

A port type definition is represented by a blue box filled in white.

9.3.5.4 XML representation

```
<xsd:complexType name="PortTypeType">
   <xsd:sequence>
     <xsd:element name="PortElement" type="PortElementType" />
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" />
 </xsd:complexType>
 <xsd:complexType name="PortElementType">
   <xsd:attribute name="interface" type="xsd:string" />
   <xsd:attribute name="kind" type="xsd:string" />
   <xsd:attribute name="name" type="xsd:string" />
 </xsd:complexType>
```

9.3.5.5 IDL equivalent syntax

A port type definition is represented by an IDL **port type**. A port element is either represented by an IDL **provides** or **uses**, depending on whether the associated interface is provided or required.

A UCM port definition in IDL follows the following rules:

1. A UCM port definition is specified using an IDL *porttype* construct.

2. There is no means to express in IDL the port role and its parameter. These information allows to link a port to an interaction pattern. Interaction patterns are not expressed in IDL.

3. The port elements associated to the UCM port specification are specified within the porttype body. It may include a set of *provides* and *uses* statements. A provides statement expresses a provided port element and a uses statement expresses a required port element. Both statements will use or provide the interface associated to the port element.

4. If the port specification uses a template parameter, its equivalent IDL porttype must be defined within an IDL template module with as much parameters as the port specification has ones.

5. UCM ports refinement cannot be expressed in IDL.

6. IDL ports attributes has no meaning in UCM.

9.4 Nonfunctional aspects package

9.4.1 Overview

Nonfunctional aspects cover the relationship between the component business code and the execution environment. They consist of the interactions between the components and the runtime libraries that support their executions, and also the programming languages supported by the UCM tool chain.

Like interactions, nonfunctional aspects are defined in two steps. Technical aspects define general semantics. Technical policy definitions specify the exact semantics and APIs if need be.

Figure 20 provides an example of such declarations. A technical aspect named “execution_policy” is defined; it is meant to specify how components are managed by the underlying runtime. Its cardinality is “exactlyOne”, meaning that any component declaration must be associated with one technical policy that conforms with this technical aspect.

In the example, two technical policies are defined: “passive” and “active_periodic”. Technical policy “active_periodic” has a port element that provides interface “activation_itf”. This means the business code of a component that is associated with technical policy “active_periodic” will have to implement interface “activation_itf”. It also defines an execution period.

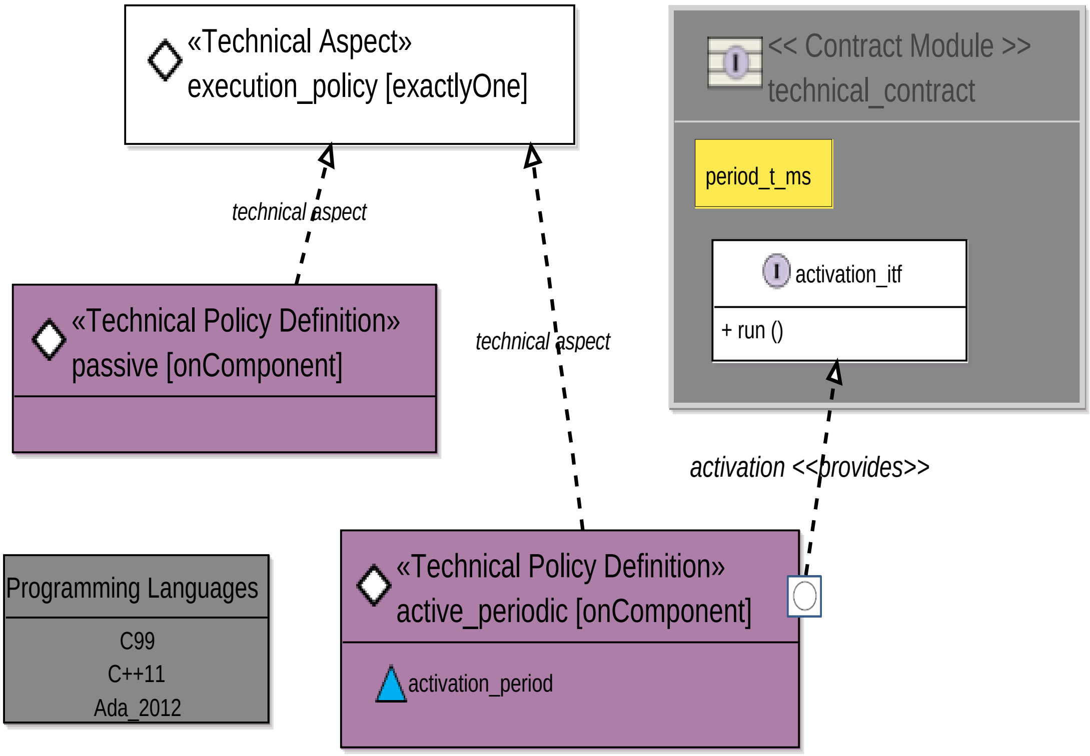

Figure 20: Example of technical policy definition

The corresponding XML representation is the following:

```
<NonfunctionalAspectModule name="policy_mod">
  <TechnicalPolicyDefinition applicability="onComponent" name="active_periodic"
                             technicalAspect="exec_asp">
    <ConfigurationParameter name="activ_period"
type="tech_contract_mod::period_t_ms"/>
    <PortElement interface="tech_contract_mod::activation_itf" kind="PROVIDED"
                 name="activation"/>
  </TechnicalPolicyDefinition>
  <TechnicalPolicyDefinition applicability="onComponent" name="passive"
                             technicalAspect="exec_asp"/>
  <ContractModule name="tech_contract_mod">
    <PrimitiveInteger kind="BYTE" name="period_t_ms"/>
    <Interface name="activation_itf">
      <Method name="run"/>
    </Interface>
  </ContractModule>
  <TechnicalAspect multiplicity="exactlyOne" name="exec_asp"/>
  <ProgrammingLanguages>
    <Language name="C99"/>
    <Language name="C++11"/>
    <Language name="Ada_2012"/>
  </ProgrammingLanguages>
</NonfunctionalAspectModule>
```

There is no IDL syntax corresponding to the technical policy definitions, as this notion does not exist in IDL. Only interfaces can be represented:

```
module policy_mod {
 module tech_contract_mod {
  typedef octet period_t_ms;
```

```
  interface activation_itf {
   void run();
  }
 }
}
```

9.4.2 Nonfunctional aspect module

The main entities of the nonfunctional aspects package are illustrated on figure 21. Technical aspects correspond to abstract notions (e.g. component execution policy). Technical policy definitions are the actual means to specify the nonfunctional aspect that will be managed by containers. They can define APIs (as in the example) and can have configuration parameters.

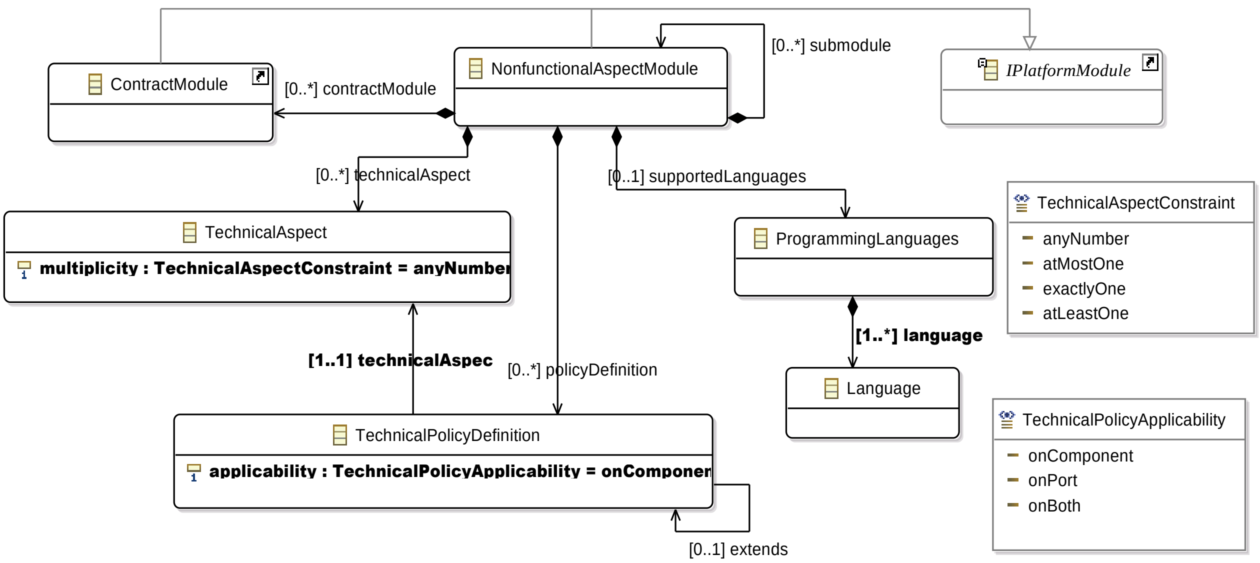

Figure 21: Main classes of UCM technical policies package

9.4.2.1 NonfunctionalAspectModule (IPlatformModule)

A nonfunctional aspect module gathers the declarations of technical policies and programming languages the platform supports. It can contain submodules in order to create hierarchical declarations. It can also contain contract modules for contracts that are associated with the technical policies.

- policyDefinition: TechnicalPolicyDefinition [0…*]
- submodule: TechnicalPolicyModule [0…*]
- contractModule: ContractModule [0…*]
- technicalAspect: TechnicalAspect [0…*]
- supportedLanguages: ProgrammingLanguages [0…1]

9.4.3 Technical policies

9.4.3.1 TechnicalAspect (INamed)

A technical aspect defines an abstract nonfunctional concept that shall be specified by a technical policy definition.

- multiplicity: TechnicalAspectConstraint [1]

Enumerated type TechnicalAspectConstraint defines the possible multiplicity of technical policies. Four possibilities are defined: anyNumer, atMostOne, exactlyOne and atLeastOne.

9.4.3.2 TechnicalPolicyDefinition (INamed, IConfigurable)

A technical policy definition specifies a capability of the container, either provided to components or enforced by the container. It actually represents any kind of nonfunctional feature managed at container level or expected from the component.

- portElement: PortElement [0…*]
- technicalAspect: TechnicalAspect [1]
- applicability: TechnicalPolicyApplicability [1]
- extends: TechnicalPolicyDefinition [0…1]

Like a connector definition, a technical policy definition must be recognized and understood by a UCM framework to be correctly interpreted and processed. Field portElement specifies possible APIs either provided to or required from the component. Internal APIs will complement the component API.

A technical policy definition can have configuration parameters to specify nonfunctional settings (e.g. execution period).

A technical policy definition can extend another one. In this situation, the technical policy definition inherits the port elements and configuration parameters defined in its ancestors. Redefinitions are forbidden.

Enumerated type TechnicalPolicyApplicability defines the valid associations of a technical policy. Three values are defined: onComponent, onPort and onBoth.

A technical policy can thus be legally associated with a component, or with one or several ports of a component. Value “onBoth” means the technical policy definition can be associated with zero or more ports. A technical policy definition meant to be associated with a component usually corresponds to some technical capability managed by the container (e.g. a periodic component execution with the associated API, or a passive execution. In the later case, the container does actually nothing). A technical policy meant to be associated with ports typically corresponds to port interceptions. A technical policy that can apply both to ports and components is likely to have different usages.

9.4.3.3 Graphical representation

Technical aspects are represented by white boxes, with a diamond icon. Technical policy definitions are represented by purple boxes (like connector definitions) with a diamond icon.

9.4.3.4 XML representation

```
<xsd:complexType name="NonfunctionalAspectModuleType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="TechnicalPolicyDefinition"
                 type="TechnicalPolicyDefinitionType" />
    <xsd:element name="ContractModule" type="ContractModuleType" />
    <xsd:element name="TechnicalAspect" type="TechnicalAspectType" />
    <xsd:element name="ProgrammingLanguages" type="ProgrammingLanguagesType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="TechnicalAspectType">
  <xsd:attribute name="multiplicity" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
```

```
<xsd:complexType name="TechnicalPolicyDefinitionType">
  <xsd:sequence>
    <xsd:element name="ConfigurationParameter" type="ConfigurationParameterType"
/>
    <xsd:element name="PortElement" type="PortElementType" />
  </xsd:sequence>
  <xsd:attribute name="applicability" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="technicalAspect" type="xsd:string" />
</xsd:complexType>
```

9.4.3.5 IDL equivalent syntax

IDL does not have corresponding keywork for technical policy definition. A nested contract module is represented by IDL **module**.

9.4.4 Supported programming languages

The programming languages supported by a given UCM framework are listed in nonfunctional aspect modules. UCM frameworks should ship with a technical policy package that contains the list of the language they support.

9.4.4.1 ProgrammingLanguages

Programming languages are a list of language declarations.

- languages: Language [1…*]

9.4.4.2 Language (INamed)

Field identifier of class Language should be the actual name of the language (e.g. “C”, “Ada”, etc.).

9.5 Components package

UCM components contain the business logic of the application. They are designed by users while interactions and nonfunctional aspects are designed by platform providers.

9.5.1 Overview

Components hold the functional part of UCM architectures. The ucm_components package focuses on the definition of these components as reusable blocks. The UCM standard makes a clear distinction between the specification of functional blocks (called **component types**) and the specification of how those blocks should behave internally (called **component implementations**).

Component types aggregate the functional contracts offered by the component to the rest of the application. Functional contracts consist of interaction patterns (defined by ucm_interactions packages, see section 9.3) and associated data or service (defined by ucm_types packages, see section 9.2. They are specified by **ports**.

Component implementations describe the internal structures that correspond to component types. A given component type can have several implementations. Component implementations can be either atomic or composite. Atomic component implementations encapsulate behaviors (i.e. source code) while composite component implementations contain subcomponents, thus allowing for architecture breakdown.

Figure 22 gives an example of component declarations. It shows three component definitions: “C1_producer”, “C2_consumer” and “C3_transmitter”. Ports of component definitions reference port API definitions declared in an interaction module (§ 9.3.1). Port APIs can used by several components.

An atomic component implementation is associated to each component definition: “C1_impl1”, “C2_impl1” and “C3_impl1”. Component implementation “C1_impl1” is associated with a technical policy “active”; component implementations “C2_impl1” and “C3_impl1” are associated with technical policy “passive” (§ 9.4.1). Consequently, the business code of C1_impl1 will use interface “activation_itf”.

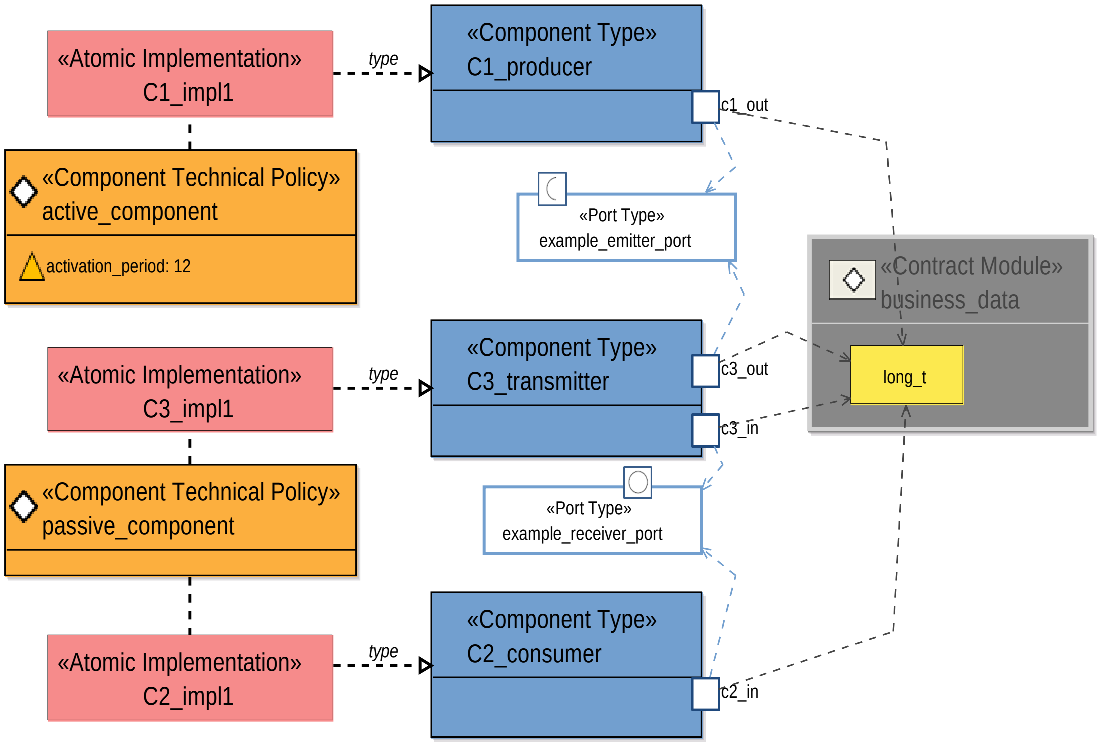

Figure 22: Example of component declarations

The XML representation for the contract module is the following:

```
<ContractModule name="business_data">
  <PrimitiveInteger kind="LONG" name="long_t"/>
</ContractModule>
```

The corresponding IDL representation is the following:

```
module business_data {
 typedef long long_t;
}
```

The corresponding XML representation for the component module is the following:

```
<ComponentModule name="comp">
  <AtomicComponentImplementation language="C++11" name="C1_impl1"
                                 type="C1_producer">
    <TechnicalPolicy name="active_comp"/>
  </AtomicComponentImplementation>
  <AtomicComponentImplementation language="C++11" name="C2_impl1"
                                 type="C2_consumer">
    <TechnicalPolicy name="passive_comp"/>
  </AtomicComponentImplementation>
  <AtomicComponentImplementation language="C++11" name="C3_impl1"
                                 type="C3_transmitter">
    <TechnicalPolicy name="passive_comp"/>
```

```
  </AtomicComponentImplementation>
  <ComponentTechnicalPolicy name="active_comp"
          technicalPolicyDefinition="::policy_mod::active_periodic">
    <ManagedComponent name="C1_impl1"/>
    <ConfigurationParameterValue
       configurationParameter="::policy_mod::active_periodic.activ_period"
       value="12"/>
  </ComponentTechnicalPolicy>
  <ComponentTechnicalPolicy name="passive_comp"
       technicalPolicyDefinition="::policy_mod::passive">
    <ManagedComponent name="C2_impl1"/>
    <ManagedComponent name="C3_impl1"/>
  </ComponentTechnicalPolicy>
  <ComponentType name="C1_producer">
    <Port name="c1_out">
      <PortTypeSpec type="::intr_mod::emit_prt">
        <AbstractTypeBinding
             abstractType="::intr_mod::intr_contr_mod::data_type_t"
             actualType="::business_data::long_t"/>
      </PortTypeSpec>
    </Port>
  </ComponentType>
  <ComponentType name="C2_consumer">
    <Port name="c2_in">
      <PortTypeSpec type="::intr_mod::receive_prt">
        <AbstractTypeBinding
             abstractType="::intr_mod::intr_contr_mod::data_type_t"
             actualType="::business_data::long_t"/>
      </PortTypeSpec>
    </Port>
  </ComponentType>
  <ComponentType name="C3_transmitter">
    <Port name="c3_in">
      <PortTypeSpec type="::intr_mod::receive_prt">
        <AbstractTypeBinding
             abstractType="::intr_mod::intr_contr_mod::data_type_t"
             actualType="::business_data::long_t"/>
      </PortTypeSpec>
    </Port>
    <Port name="c3_out">
      <PortTypeSpec type="::intr_mod::emit_prt">
        <AbstractTypeBinding
            abstractType="::intr_mod::intr_contr_mod::data_type_t"
            actualType="::business_data::long_t"/>
      </PortTypeSpec>
    </Port>
  </ComponentType>
</ComponentModule>
```

The IDL syntax for component types is the following:

```
module comp {
 alias ::intr_mod<business_data::long_t> my_intr_mod;
 component C1_producer {
  port my_intr_mod::emit_prt c1_out;
 }
 component C2_consumer {
  port my_intr_mod::receive_prt c2_in;
 }
```

```
component C3_transmitter {
  port my_intr_mod::emit_prt c3_out;
  port my_intr_mod::receive_prt c3_in;
 }
}
```

Component implementations and technical policies cannot be represented in IDL.

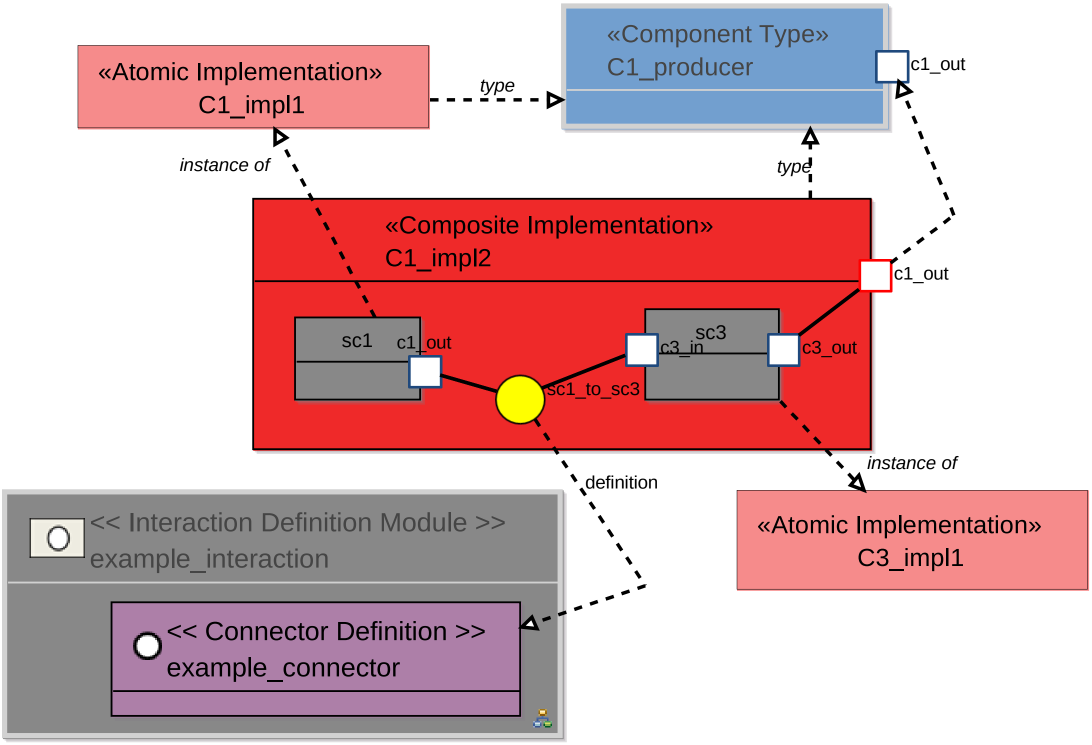

Figure 23: Example of composite component implementation

Figure 23 shows the definition of a composite component implementation “C1_impl2”, associated with C1_producer. Composite component implementations have subcomponents named “assembly parts”. On figure 23, we see that C1_impl2 contains two subcomponents: sc1 is an instance of C1_impl1, and sc3 is an instance of C3_impl1. These two instances are connected using connector “example_connector” (§ 9.3.1). Port c3_p2 of instance sc3 is connected to port c1_p1 of C1_impl1 by a port delegation: such a delegation is possible because both ports reference the same port API.

9.5.2 Component Module

Component modules contain the different declarations related with the business entities of architectures: components with their ports, component implementations with their features or subcomponents.

The main entities of the component package are illustrated on figure 24.

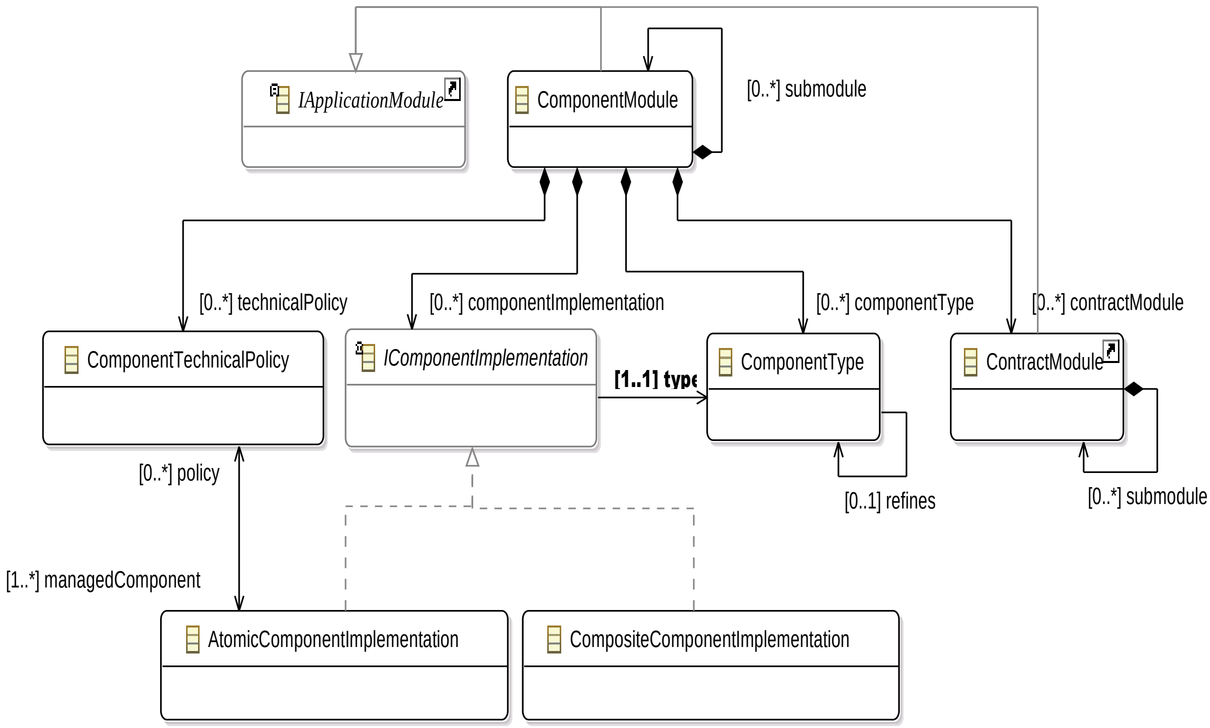

Figure 24: Main classes involved in UCM component package

9.5.2.1 ComponentModule (IApplicationModule)

The ComponentModule class is meant to contain all component definitions. It can contain submodules in order to create hierarchies. It can also contain contract modules for data type declarations that are directly related with components.

- submodule: ComponentModule [0…*]
- contractModule: ContractModule [0…*]
- componentType: ComponentType [0…*]
- componentImplementation: IComponentImplementation [0…*]
- technicalPolicy: ComponentTechnicalPolicy [0…*]

9.5.2.2 IComponent (INamed, IAnnotable)

IComponent is an abstract class that represents any kind of component declaration (either component definition of component implementation). It is meant to serve as a common ancestor for all these declarations.

All kinds of component declarations inherit from IComponent. Components can have annotations to decorate the functional declarations.

9.5.2.3 IComponentImplementation (IComponent)

Abstract class IComponentImplementation represents any kind of component implementation. The UCM standard defines two concrete classes that extend this class: AtomicComponentImplementation (§ 9.5.4) and CompositeComponentImplementation (§ 9.5.5).

- type: ComponentType [1]

9.5.3 Component types and ports

Component definitions are the functional contracts of components: they define component possible interactions.

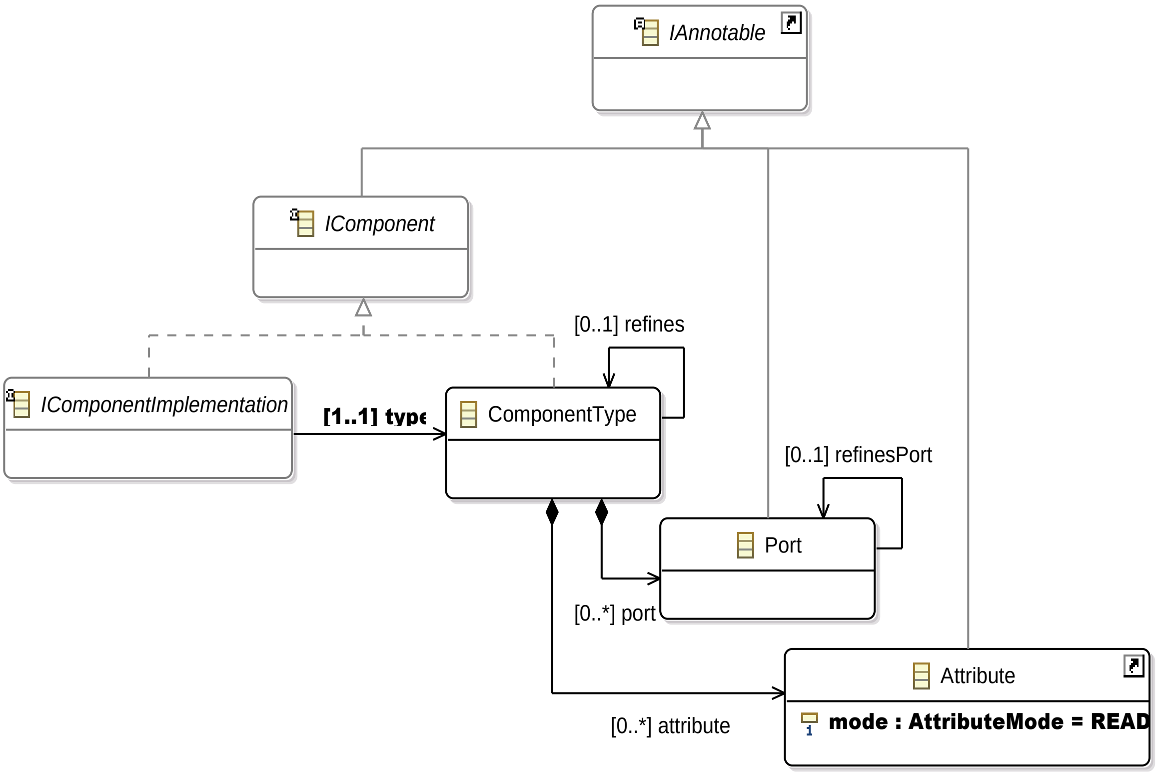

Figure 25: UCM component types

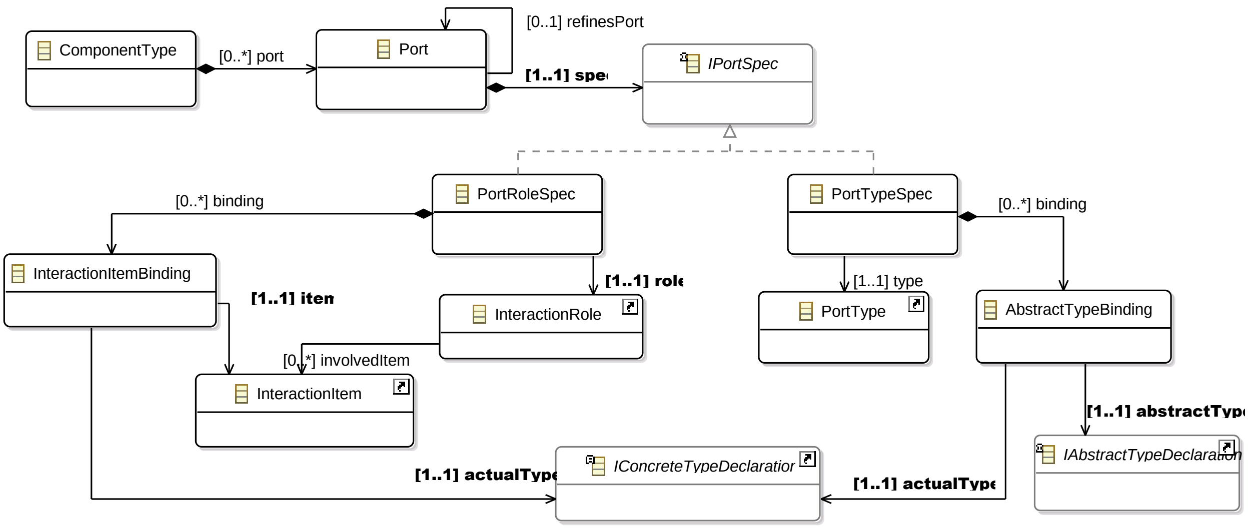

Figure 26: UCM component ports

9.5.3.1 ComponentType (IComponent)

Component definitions specify the functional contracts that enable interactions between a given component and the rest of the application.

- port: Port [0…*]
- attribute: Attribute [0…*]
- refines: ComponentType [0…1]

Attribute definition is imported from the ucm_types package (§ 9.2.7.4). Attributes are used to specify functional parameters that could be handled by the business code inside components. Other components cannot see them, but the container can.

A component type can refine another component type. In this situation, the component type inherits the ports and attributes of its ancestor. It is important to note that component refinement is different from the subtyping mechanism of object-oriented programming. In a given architecture, a component type cannot be used in place of one of its ancestors. The refinement relationship is thus an inheritance relationship but not a subtyping relationship.

A port of a given component type cannot have the same name as a port of its ancestor, unless it refines it (§ 9.5.3.2). An attribute cannot have the same name as an attribute of an ancestor of the component. In order to prevent name conflicts between ancestors, a given component type can refine at most one component definition.

9.5.3.2 Port (INamed, IAnnotable)

Ports specify component interaction points. They are associated with a port specification (§ 9.5.3.3).

- spec: IPortSpec [1]
- refinesPort: Port [0…1]

As IPortSpec can correspond either to a port type specification or to a port role definition, a port is defined either by an explicit set of APIs or simply by a role. Consequently, a component definition is not necessarily a set of APIs: it can be less precise than that, which allows iterative refinement when designing architectures.

The refinesPort field is used in case of port refinement. The refined port must be contained in an ancestor component definition. It does not need to have the same name as the refining port. A given port can be refined by several ports at a time; it means the port refinement actually leads to decomposition into several ports.

Ports can have properties. Properties can be typically used to specify assumptions made by the component in order to execute properly. For example, a property could be associated with a component port to indicate an expected rate for data inputs.

9.5.3.3 IPortSpec

Abstract class IPortSpec is referenced by component ports. It enables UCM frameworks to provide additional, framework-specific ways to define UCM ports specifications. The UCM standard defines two concrete classes that inherit this class: PortRoleSpec and PortTypeSpec.

9.5.3.4 PortRoleSpec (IPortSpec)

A PortRoleSpec references an interaction role and specifies the binding of the interaction items with actual type declarations.

- role: InteractionRole [1]

- binding: InteractionItemBinding [0…*]

Port role specifications can be used to specify component ports in the early stages of the architecture definition process. Referencing a role allows the specification of components with respect to interaction patterns, that is, with respect to an interaction logic, rather than actual API.

9.5.3.5 InteractionItemBinding

Class InteractionItemBinding defines the binding between an item of an interaction pattern and an actual type declaration (either data type or interface).

- item: InteractionItem [1]
- actualType: IConcreteTypeDeclaration [1]

The class ITypeDeclaration is defined in the ucm_types package, and corresponds to any type declaration.

9.5.3.6 PortTypeSpec (IPortSpec)

A PortTypeSpec is similar to a port role specification, except that it references a port type instead of an interaction role.

- type: PortType [1]
- binding: AbstractTypeBinding [0…*]

Port type specifications are used to completely specify component ports, as they reference a port type, that is, an API.

9.5.3.7 AbstractTypeBinding

Class AbstractTypeBinding defines the binding between an abstract type used in the port type referenced by the component port and an actual type declaration (either data type or interface).

- abtractType: IAbstractTypeDeclaration [1]
- actualType: ITypeDeclaration [1]

9.5.3.8 Graphical representation

Component types are represented by blue boxes, with no icon.

9.5.3.9 XML representation

```
<xsd:complexType name="ComponentTypeType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="Port" type="PortType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="PortType">
  <xsd:sequence>
    <xsd:element name="PortTypeSpec" type="PortTypeSpecType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="PortTypeSpecType">
  <xsd:sequence>
    <xsd:element name="AbstractTypeBinding" type="AbstractTypeBindingType" />
  </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="AbstractTypeBindingType">
  <xsd:attribute name="abstractType" type="xsd:string" />
  <xsd:attribute name="actualType" type="xsd:string" />
</xsd:complexType>
```

9.5.3.10 IDL equivalent syntax

A UCM component type may be expressed as an IDL **component.** The IDL component must have extended port declarations only, corresponding to the UCM ports definitions. It is not allowed to declare basic facet and/or receptacle ports. These extended ports types are defined in an external IDL file provided by the platform provider. Any UCM component attribute is expressed as an IDL attribute within the IDL component. There is no means to express default values for the components attributes in IDL. Component refinement id expressed by IDL component inheritance. Similarly to a UCM component that can refine one other component only, an IDL component can inherit from one other component only.

A component port is declared within the IDL component by an extended **port**. If the port definition is included in a template module, this module must be instantiated first using the IDL **alias** keyword. The port definition is then referred to using the module actual instance.

9.5.4 Atomic component implementations and technical policies

Atomic component implementations correspond to deployable entities that encapsulate behavior. As atomic component implementations are the actual holders for business logic, they are controlled by containers.

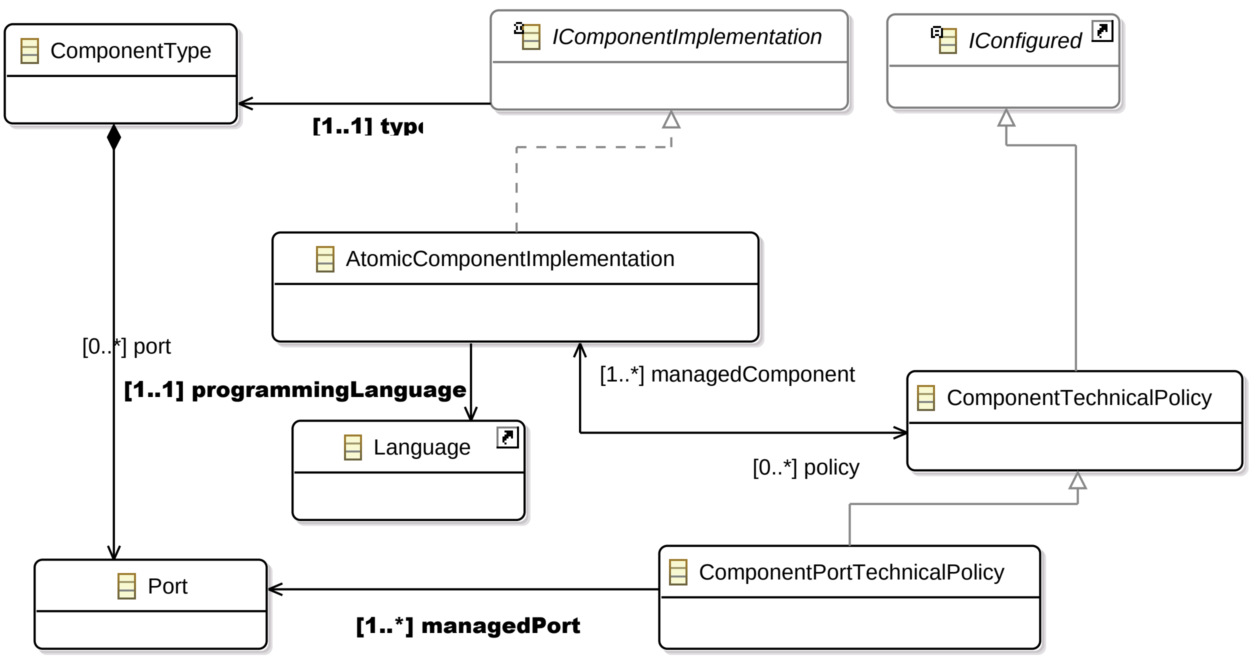

Figure 27: UCM atomic component implementations

Technical policies are associated with atomic component implementations to specify interactions with containers.

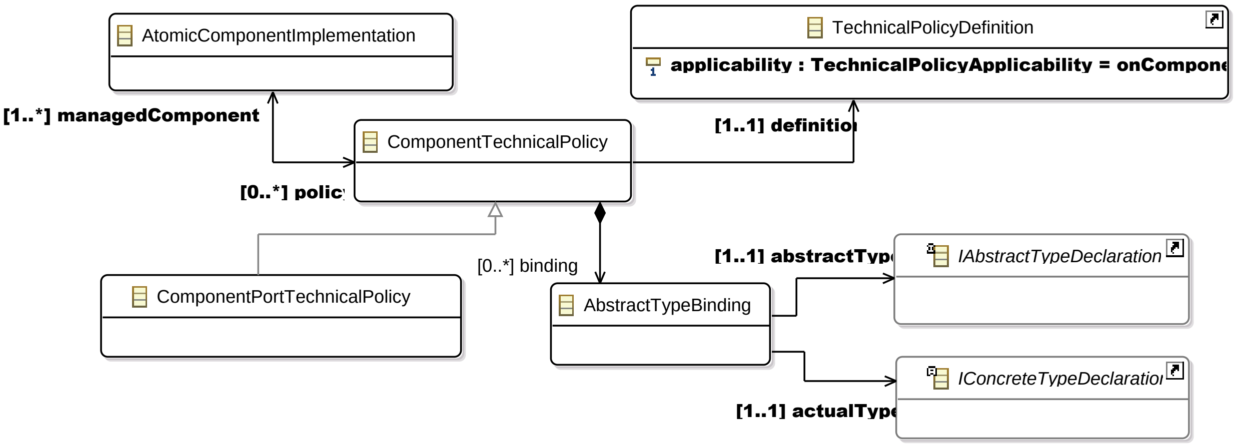

Figure 28: UCM technical policies

9.5.4.1 AtomicComponentImplementation (IComponentImplementation)

Class AtomicComponentImplementation represent actual business logic.

- programmingLanguage: Language [1]
- policy: ComponentTechnicalPolicy [0…*]

Field programmingLanguage indicates the programming language used to write the implementation code. It references a language among those defined in a technical policy definition module (§ 9.4.4).

An atomic component implementation can be associated with technical policies to specify interactions with or configurations of the component container.

9.5.4.2 ComponentTechnicalPolicy (INamed, IConfigured)

Component technical policies apply to atomic component implementations. They thus materialize the application of a technical policy to one or several component implementations.

- managedComponent: AtomicComponentImplementation [1…*]
- definition: TechnicalPolicyDefinition [1]
- binding: AbstractTypeBinding [0…*]

A given component technical policy can be applied to several atomic component implementations at a time. This can be used to share a given technical service between several components (e.g. a lock service). Configuration parameters defined in the corresponding technical policy definition can receive values.

Component technical policies can only reference a technical policy definition the applicability of which is “onComponent” or “onBoth” (§ 9.4.3.2).

Like port type specifications, a component technical policy can have type bindings, to be used if the port elements of the technical policy definition rely on abstract type declarations.

9.5.4.3 ComponentPortTechnicalPolicy (ComponentTechnicalPolicy)

Component port technical policies have the same role as component technical policies, but they apply to ports of atomic component implementations. They have the following additional field:

- managedPort: Port [1…*]

A given component port technical policy can be associated with several ports of several components at a time.

Component port technical policies can only reference a technical policy definition the applicability of which is “onPort” or “onBoth” (§ 9.4.3.2).

9.5.4.4 Graphical representation

Atomic component implementation are represented by light red boxes, with no icon.

9.5.4.5 XML representation

```
<xsd:complexType name="AtomicComponentImplementationType">
  <xsd:sequence>
    <xsd:element name="TechnicalPolicy" type="TechnicalPolicyType" />
  </xsd:sequence>
  <xsd:attribute name="language" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="type" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="TechnicalPolicyType">
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
```

9.5.4.6 IDL equivalent syntax

There is no IDL syntax for atomic component implementations and technical policies.

9.5.5 *Composite Component Implementations*

The definition of a composite component covers its internal decomposition into subcomponents and connections between the ports of these subcomponents. Subcomponents are named AssemblyPart. A composite implementation also contains port delegations to delegate its ports to ports of subcomponents.

An AssemblyPart references an IComponent. This means an assembly part can reference either a component definition or a component implementation.. The normal usage is to reference a component implementation to create complete architectures. However, the UCM standard allows create assembly parts that reference component types in order to support high-level architecture designs.

Connections have ConnectionEnd elements, which are connected to an AssemblyPart and a Port of the corresponding ComponentDefinition of the AssemblyPart.

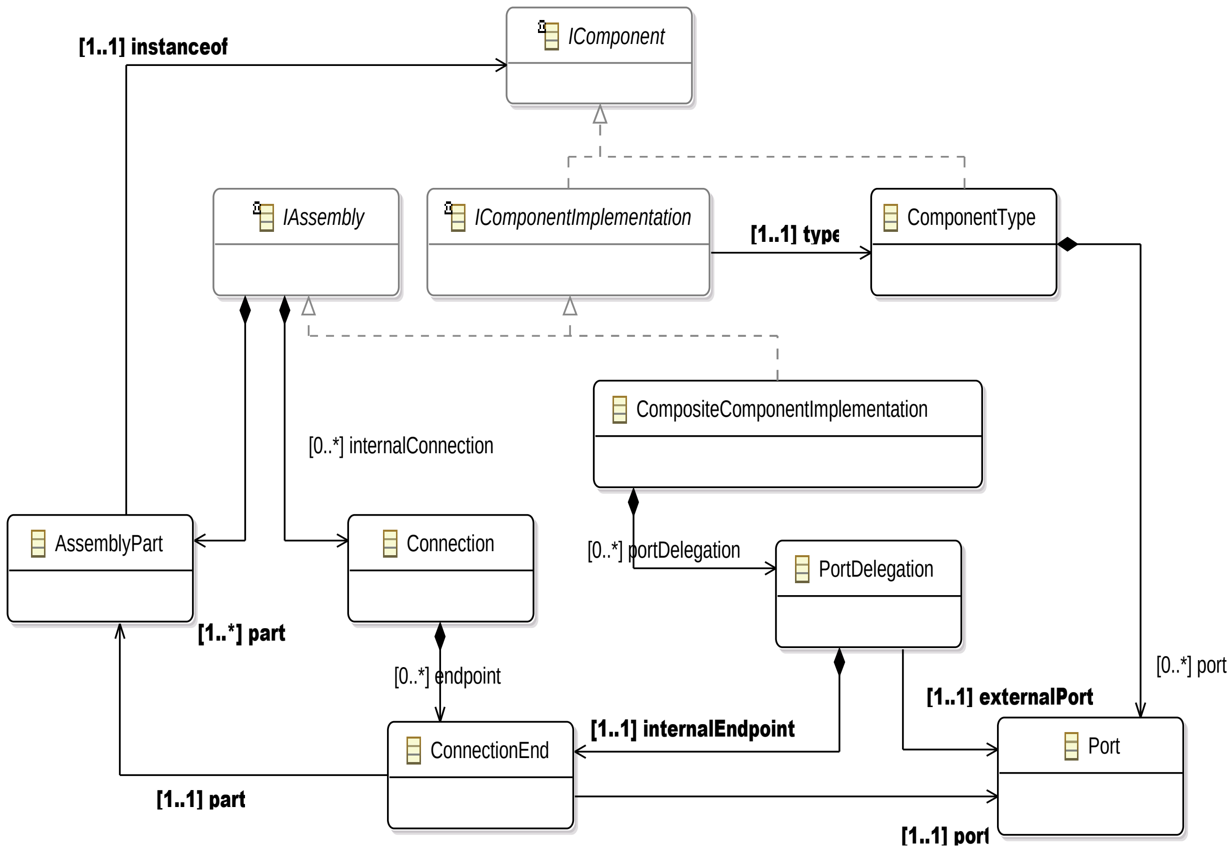

Figure 29: UCM composite component implementations

9.5.5.1 IAssembly

Abstract class IAssembly defines assemblies. In the UCM standard, this concept is only extended by the CompositeComponentImplementation class, but meta-model extensions can reuse it to describe deployments.

- part: AssemblyPart [1…*]
- internalConnection: Connection [0…*]

Parts are sub-elements of the assembly.

9.5.5.2 CompositeComponentImplementation (IComponentImplementation, IAssembly)

A composite component implementation contains parts, internal connections and port delegations.

- portDelegation: PortDelegation [0…*]

9.5.5.3 AssemblyPart (INamed)

An assembly part is a sub-component of an assembly. It references a component declaration (either component definition or component implementation).

- instanceof: IComponent [1]

Assembly parts can either reference a component type or a component implementation. Referencing component types enables the definition of composite implementation in the early stages of the architecture definition process.

9.5.5.4 Connection (INamed)

Connections are instances of connector definitions or interaction pattern definitions. They are used to connect ports of sub-components.

- endpoint: ConnectionEnd [0…*]
- connectionDefinition: IInteractionDefinition [1]

Connections can reference either a connector or an interaction pattern. The UCM standard thus enables early design of architectures, where the exact interaction mechanisms are not yet set.

9.5.5.5 ConnectionEnd (INamed)

Connection ends connect connections to ports of assembly parts.

- part: AssemblyPart [1]
- port: Port [1]

9.5.5.6 PortDelegation

Composite port delegations allow the complete delegation of a port of a composite component implementation to a port of a sub-component. The definitions of both ports must be the same.

- externalPort: Port [1]
- internalEndPoint: ConnectionEnd [1]

The external port belongs to the component type of the composite component implementation. The internal end point references the port and the corresponding the sub-component. Unlike connections, port delegations are not associated with a connector definition or an interaction pattern: they simply bind the external port to a port of a subcomponent.

9.5.5.7 Graphical representation

Composite component implementations are represented by red boxes that contain subcomponents and connections. Subcomponents (assembly parts) are represented by gray boxes. Connections are represented by yellow discs.

9.5.5.8 XML representation

```
<xsd:complexType name="CompositeComponentImplementationType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="Part" type="PartType" />
    <xsd:element name="Connection" type="ConnectionType" />
    <xsd:element name="PortDelegation" type="PortDelegationType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="type" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="PortDelegationType">
  <xsd:sequence>
    <xsd:element name="ConnectionEnd" type="ConnectionEndType" />
  </xsd:sequence>
  <xsd:attribute name="internalPart" type="xsd:string" />
  <xsd:attribute name="internalPort" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="port" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="ConnectionEndType">
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="part" type="xsd:string" />
```

```
  <xsd:attribute name="port" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="ConnectionType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="ConnectionEnd"
type="ConnectionEndType" />
  </xsd:sequence>
  <xsd:attribute name="definition" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="ConnectionEndType">
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="part" type="xsd:string" />
  <xsd:attribute name="port" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="PartType">
  <xsd:attribute name="instanceOf" type="xsd:string" />
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
```

9.5.5.9 IDL equivalent syntax

There is no IDL syntax for composite component implementations

10. Specification of UCM platform capabilities

This section describes the standard specifications of UCM platforms. These specifications define the semantics and APIs for the component execution models, the component interaction models and the technical policies implemented by containers.

These capabilities are declared in UCM interaction and nonfunctional aspect modules and associated contract modules. The corresponding UCM models are provided in machine-readable document mars/16-05-07.

10.1 Core UCM specifications (Normative, mandatory)

This section explains the capabilities that any UCM platform has to provide in order to conform with the core UCM standard. The connector and technical policy definitions have no configuration parameters: they only define APIs to remain portable. UCM frameworks should provide more detailed definitions by extending these, adding configuration parameters that correspond to the targeted platform capabilities.

10.1.1 Restrictions on data type declarations

Native types (§ 9.2.5.3) can be used to manipulate framework-dependent data, and thus may prevent code portability. The usage of native types is therefore not in the scope of the core UCM specifications. Frameworks that are compliant with core UCM specifications need not support them.

Attribute declarations in interfaces (§ 9.2.7.4) represent access methods rather than actual data. To avoid ambiguities, they are not part of the core UCM specifications. However, attribute declarations in components are supported.

10.1.2 Interaction return codes

Interactions should notify the business code whether communications succeeded or failed. The core UCM specifications define three basic return code for this.

```
<ContractModule name="return_codes">
  <Enumeration indexType="comm_ecode_enumerator_t" name="comm_ecode">
    <Enumerator name="ok" value="0"/>
    <Enumerator name="internal_error" value="1"/>
    <Enumerator name="comm_error" value="2"/>
  </Enumeration>
  <PrimitiveType kind="BYTE" name="comm_ecode_enumerator_t"/>
</ContractModule>
```

Value “ok” corresponds to normal behavior, where data is correctly transmitted. Value “internal_error” corresponds to an error inside the connector. Value “comm_error” corresponds to an error during the transmission (e.g. a network error).

10.1.3 Standard component execution policies

The component execution model is managed by the component_execution_policy technical aspect. A UCM component must have exactly one execution model technical policy. The UCM standard defines four technical policies: protected self-executing, protected active, protected passive and unprotected passive.

10.1.3.1 Specifications

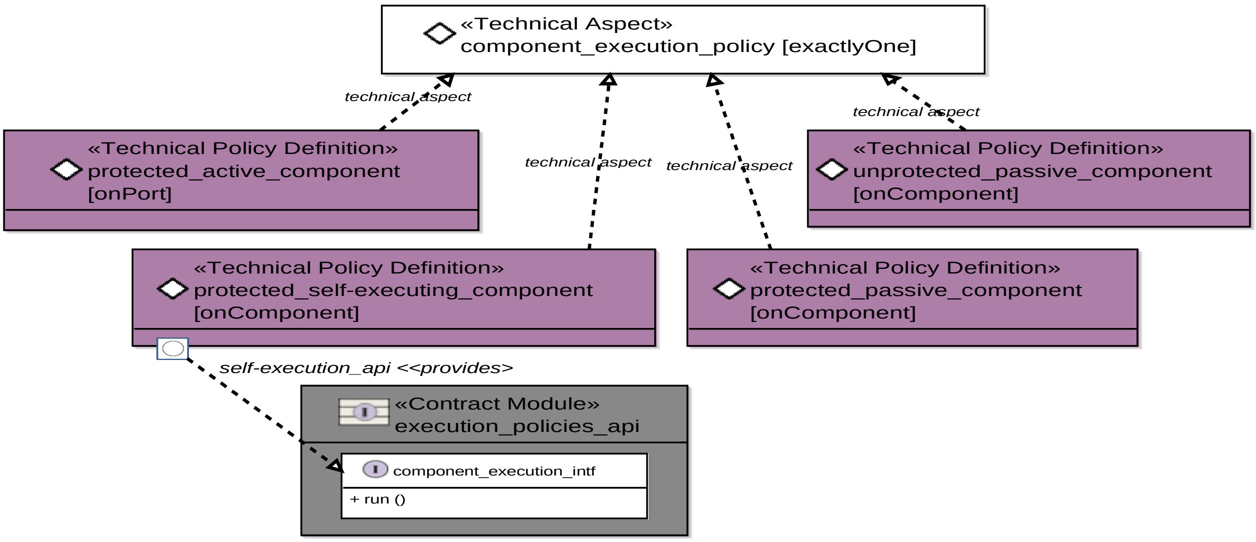

Figure 30: Standard component execution model

The corresponding declarations is shown in XML syntax below.

```
<NonfunctionalAspectModule name="component_execution_policies">
  <TechnicalPolicyDefinition applicability="onComponent"
                             name="protected_self-executing_component"
                             technicalAspect="component_execution_policy">
    <PortElement interface="execution_policies_api::component_execution_intf"
                 kind="PROVIDED" name="self-execution_api"/>
  </TechnicalPolicyDefinition>
  <TechnicalPolicyDefinition applicability="onComponent"
                             name="unprotected_passive_component"
                             technicalAspect="component_execution_policy"/>
  <TechnicalPolicyDefinition applicability="onComponent"
                             name="protected_passive_component"
                             technicalAspect="component_execution_policy"/>
  <TechnicalPolicyDefinition applicability="onPort"
                             name="protected_active_component"
                             technicalAspect="component_execution_policy"/>
  <ContractModule name="execution_policies_api">
    <Interface name="component_execution_intf">
      <Method name="run"/>
    </Interface>
  </ContractModule>
  <TechnicalAspect multiplicity="exactlyOne" name="component_execution_policy"/>
</NonfunctionalAspectModule>
```

These four technical policies must be supported by any UCM platform. Additional, non standard technical policies may be provided by platforms.

10.1.3.2 Semantics

The execution of a self-executing component is triggered by its container by calling a run() method. That is, the component is triggered by itself, without requiring any data input. The expression of the triggering conditions (e.g. the execution period in the case of a periodic trigger) is specific to each framework. A self-executing component is not reentrant.

The protected active policy applies to one or several ports. The invocation of one of these ports triggers the execution of the component. The execution is not reentrant. Like self-executing components, the execution details of active protected components (e.g. periodic or sporadic execution, exact execution resource, etc.) is not covered by the core UCM specifications; UCM framework may provide extended technical policies to manage configuration.

A passive protected component is not reentrant but does not execute by itself: it reacts to incoming calls. The container shall guarantee that the component is only executed once at a time. There is no API.

A passive component is not self-executing. Unlike other policies, it can be reentrant: several components can call it at a time. There is no API. Passive components can be compared to libraries.

10.1.4 Clock and trace service

10.1.4.1 Clock

The core UCM standard defines a technical aspect for a clock service that containers can provide to their components. A UCM component can have at most one technical policy related with the clock technical aspect. The core UCM specification defines one technical policy with an API. UCM extensions may define alternative clock technical policies.

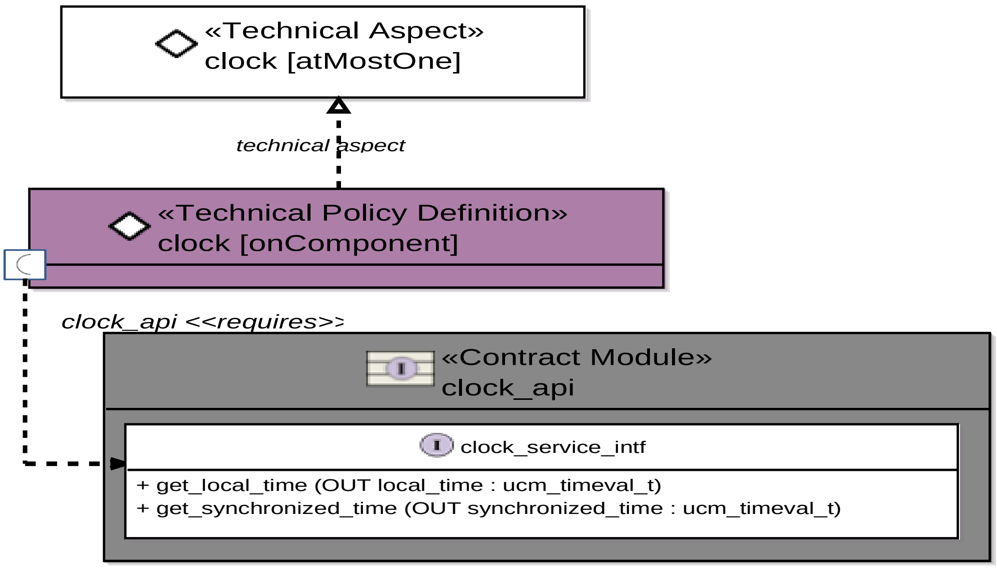

Figure 31: Standard clock service

The standard clock technical policy defines an interface that is provided by the container to the component. This interface contains two methods: get_local_time and get_synchronized_time.

Method get_local_time returns the time of the local node the component is deployed on. This is the “real” time. Method get_synchronized_time returns the global time of the whole system.

10.1.4.2 Trace

The core UCM standard defines a technical aspect for a trace service that containers can provide to their components. A UCM component can have zero or several technical policies related with the trace technical aspect. The core UCM specification defines one technical policy with an API to be manipulated by component implementation code, and one technical policy without API to be associated with ports. UCM extensions may define alternative trace technical policies.

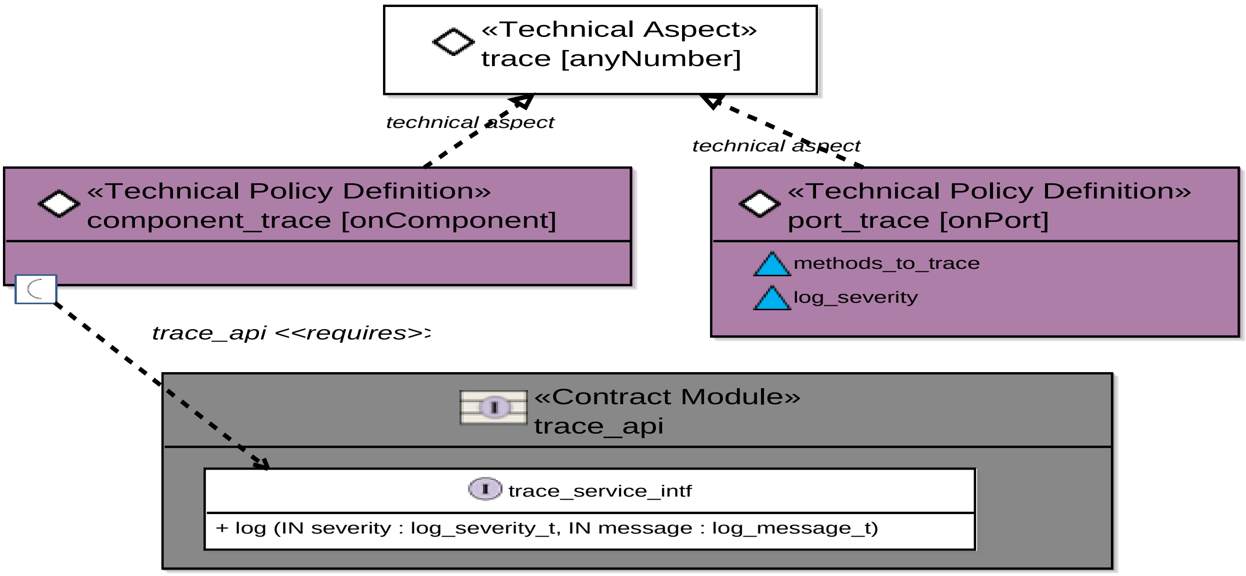

Figure 32: Standard trace service

Specifications

Definitions are gathered in a module named “container_services”, which contains two submodules: one for the clock service, the other for the trace service.

```
<NonfunctionalAspectModule name="container_services">
  <NonfunctionalAspectModule name="clock_service">
    <TechnicalAspect multiplicity="atMostOne" name="clock"/>
    <TechnicalPolicyDefinition applicability="onComponent" name="clock"
                               technicalAspect="clock">
      <PortElement interface="clock_api::clock_service_intf" kind="REQUIRED"
                   name="clock_api"/>
    </TechnicalPolicyDefinition>
```

The API for the clock service is defined in a nested module, with two methods: get_local_time and get_synchronized_time.

```
    <ContractModule name="clock_api">
      <Structure name="ucm_timeval_t">
        <!--inspired from the libC definitions-->
          <StructureField name="utv_sec" type="ucm_time_t"/>
          <StructureField name="ucm_usec" type="ucm_usecond_t"/>
      </Structure>
      <PrimitiveInteger kind="ULONG" name="ucm_time_t"/>
      <PrimitiveInteger kind="LONG" name="ucm_usecond_t"/>
      <Interface name="clock_service_intf">
        <Method name="get_local_time">
          <Parameter direction="OUT" name="local_time" type="ucm_timeval_t"/>
        </Method>
        <Method name="get_synchronized_time">
              <Parameter direction="OUT" name="synchronized_time"
                         type="ucm_timeval_t"/>
        </Method>
      </Interface>
    </ContractModule>
  </NonfunctionalAspectModule>
```

The trace service has two technical policy definitions: one that applies to ports, the other that directly applies to components. The later one defines an API to let component user code invoke the trace service.

```
  <NonfunctionalAspectModule name="trace_service">
    <TechnicalPolicyDefinition applicability="onComponent"
name="component_trace"
                               technicalAspect="trace">
      <PortElement interface="trace_api::trace_service_intf" kind="REQUIRED"
                   name="trace_api"/>
    </TechnicalPolicyDefinition>
    <TechnicalPolicyDefinition applicability="onComponent" name="port_trace"
                               technicalAspect="trace">
      <ConfigurationParameter name="methods_to_trace"
type="trace_api::method_name_t"/>
      <ConfigurationParameter name="log_severity"
type="trace_api::log_severity_t"/>
    </TechnicalPolicyDefinition>
```

The API itself is defined in a submodule.

```
    <ContractModule name="trace_api">
      <String8 name="method_name_t"/>
      <Enumeration indexType="ULONG" name="log_severity_t">
        <Enumerator name="trace" value="0"/>
        <Enumerator name="debug" value="1"/>
        <Enumerator name="info" value="2"/>
        <Enumerator name="warning" value="3"/>
        <Enumerator name="error" value="4"/>
        <Enumerator name="critical" value="5"/>
      </Enumeration>
      <PrimitiveInteger kind="BYTE" name="log_severity_enumerator_t"/>
      <String32 name="log_message_t"/>
      <Interface name="trace_service_intf">
        <Method name="log">
          <Parameter direction="IN" name="severity" type="log_severity_t"/>
          <Parameter direction="IN" name="message" type="log_message_t"/>
        </Method>
      </Interface>
    </ContractModule>
    <TechnicalAspect multiplicity="anyNumber" name="trace"/>
  </NonfunctionalAspectModule>
</NonfunctionalAspectModule>
```

10.1.5 Service based interaction

10.1.5.1 Description

Service interaction correspond to the classical client / server interaction. It involves two roles: a client and a server. There can be several clients, and there is a unique server. The definition is illustrated on figure 33.

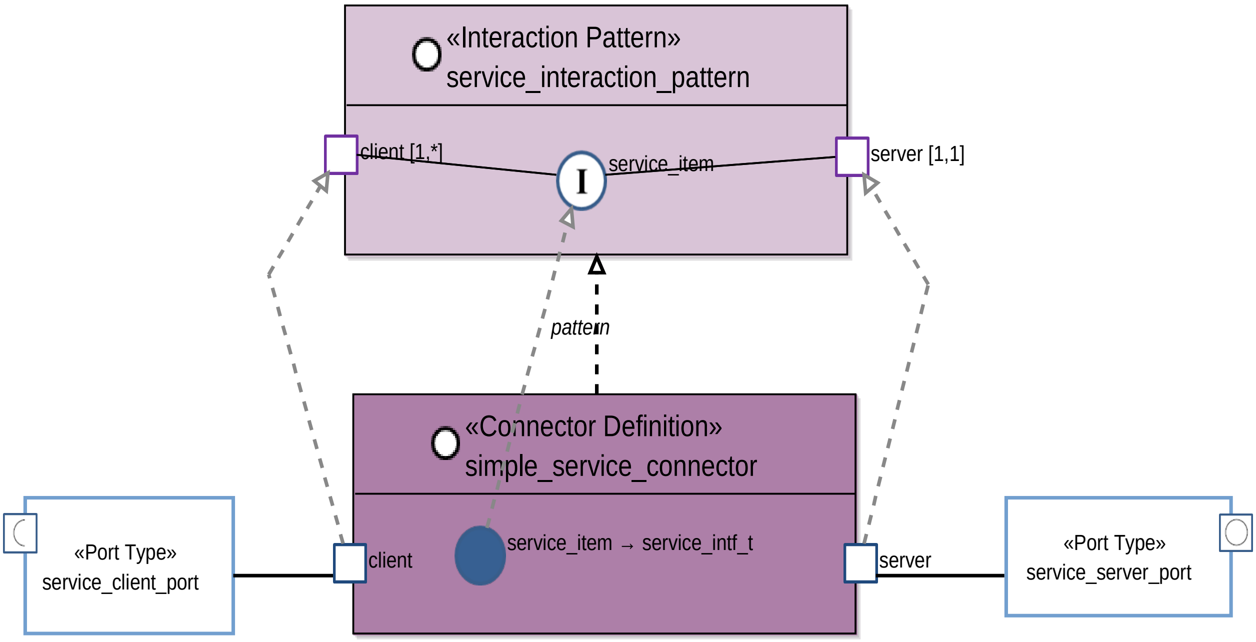

Figure 33: Service based interaction

On the server side, an interface is provided while on the client side, the same interface is required. The calls to the methods of the interface are blocking.

The two port specifications use the same interface template parameter named “service_intf_t”, as illustrated on figure 34.

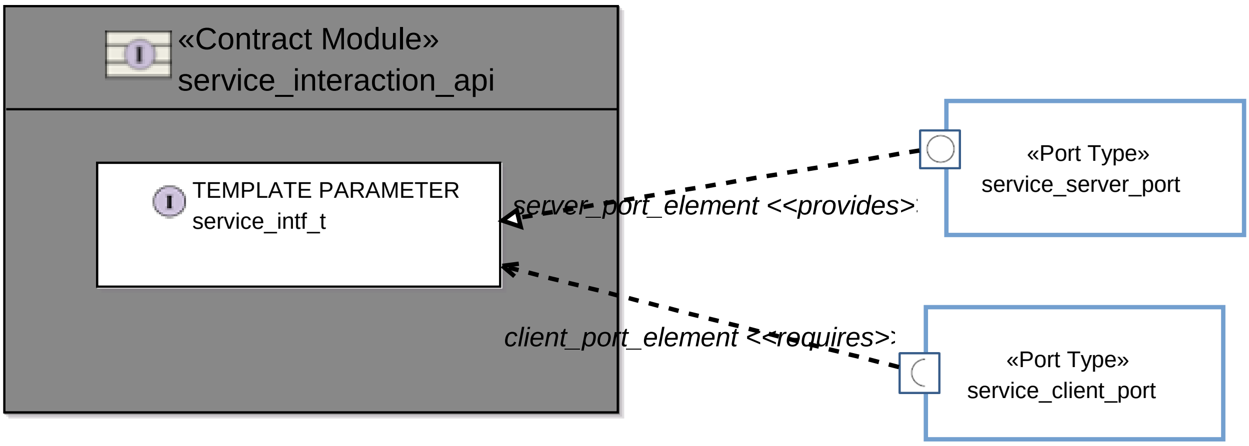

Figure 34: Port types for service based interactions

10.1.5.2 Specifications

```
<InteractionModule name="services">
   <ContractModule name="service_interaction_api">
     <InterfaceTypeTemplateParameter name="service_intf_t"/>
   </ContractModule>
   <InteractionPattern name="service_interaction_pattern">
     <InteractionRole lowerMultipliciy="1" name="client" upperMultipliciy="-1">
       <InvolvedItem item="service_intf"/>
     </InteractionRole>
```

```
    <InteractionRole lowerMultipliciy="1" name="server" upperMultipliciy="1">
      <InvolvedItem item="service_intf"/>
    </InteractionRole>
    <InteractionItem name="service_intf" nature="INTERFACE"/>
  </InteractionPattern>
  <TemplatedPort name="service_client_tport">
    <PortElement interface="service_interaction_api::service_intf_t"
                 kind="REQUIRED" name="api"/>
  </TemplatedPort>
  <TemplatedPort name="service_server_tport">
    <PortElement interface="service_interaction_api::service_intf_t"
                 kind="PROVIDED" name="api"/>
  </TemplatedPort>
  <ConnectorDefinition name="simple_service_connector"
                       pattern="service_interaction_pattern">
    <ConnectorPortDefinition name="service_client"
                             portRole="service_interaction_pattern.client"/>
    <ConnectorPortDefinition name="service_server"
                             portRole="service_interaction_pattern.server"/>
    <ItemBinding connectorItem="service_interaction_api::service_intf_t"
                 interactionItem="service_interaction_pattern.service_intf"/>
  </ConnectorDefinition>
</InteractionModule>
```

10.1.6 Message based interaction

10.1.6.1 Description

The UCM message base interaction is inspired by CCM message ports. The interaction pattern involves two roles: an emitter and a receiver. There can be several emitters and several receivers.

A standard connector is defined for this interaction pattern. The connector defines two ports: one corresponds to the emitter role, the other corresponds to the receiver role. The emitter port references a port specification named “message_emitter_port”. This port specification contains a single port element that requires interface “message_intf”. The receiver port references a port specification named “message_receiver_port”. This port specification also contains a single port element the provides the same interface.

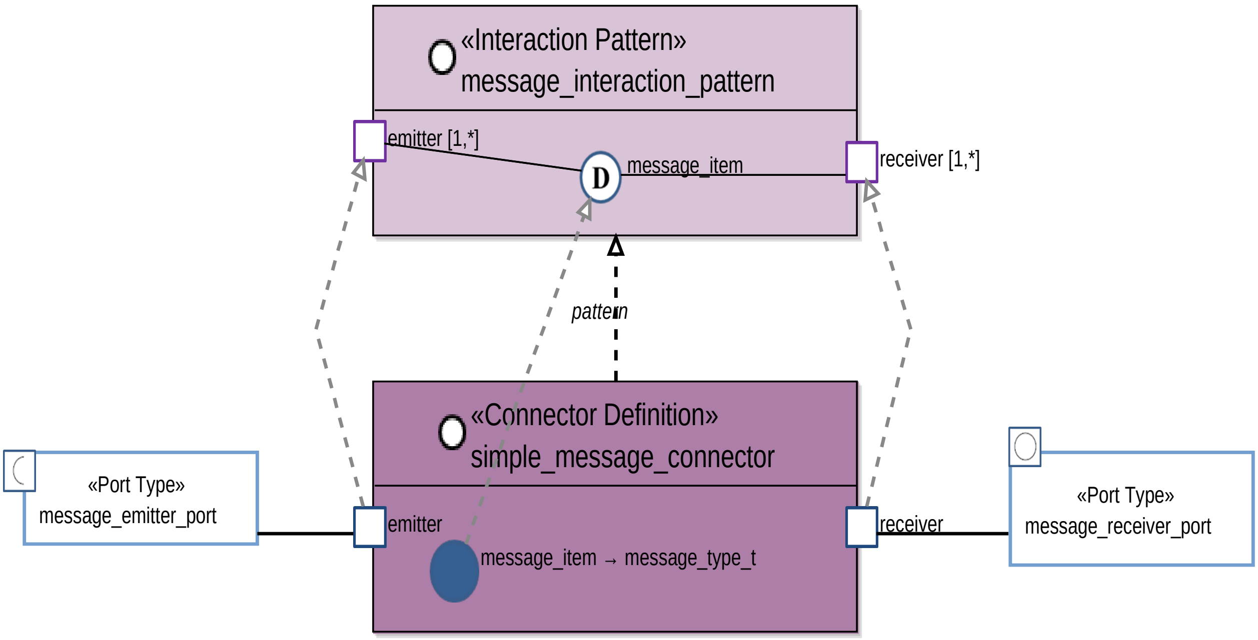

Figure 35: Message based interaction

The two port specifications use the same interface named “message_intf”. This interface has a unique method, named “push”; it takes one parameter “message”, the type of which is a data type template parameter named “message_type_t”.

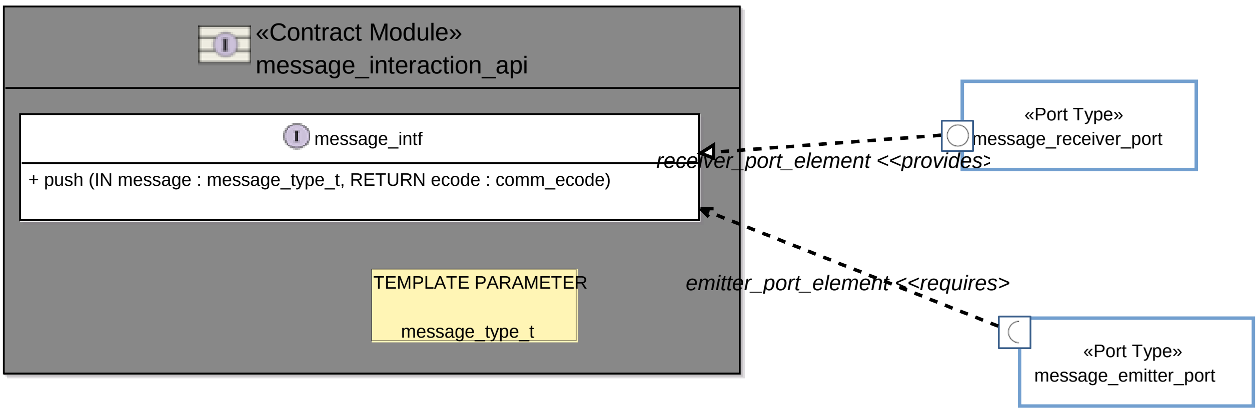

Figure 36: Port types for message based interactions

10.1.6.2 Specifications

```
<InteractionModule name="messages">
  <ContractModule name="message_interaction_api">
    <DataTypeTemplateParameter name="message_type_t"/>
    <Interface name="message_intf">
      <Method name="push">
        <Parameter direction="IN" name="message" type="message_type_t"/>
        <Parameter direction="RETURN" name="ecode"
                   type="::return_codes::comm_ecode"/>
      </Method>
    </Interface>
  </ContractModule>
```

```
  <InteractionPattern name="message_interaction_pattern">
    <InteractionRole lowerMultipliciy="1" name="emitter" upperMultipliciy="-1">
      <InvolvedItem item="message_item"/>
    </InteractionRole>
    <InteractionRole lowerMultipliciy="1" name="receiver" upperMultipliciy="-1">
      <InvolvedItem item="message_item"/>
    </InteractionRole>
    <InteractionItem name="message_item" nature="DATA"/>
  </InteractionPattern>
  <TemplatedPort name="message_emitter_port">
    <PortElement interface="message_interaction_api::message_intf"
kind="REQUIRED"
                 name="emitter_port_element"/>
  </TemplatedPort>
  <TemplatedPort name="message_receiver_port">
    <PortElement interface="message_interaction_api::message_intf"
kind="PROVIDED"
                 name="receiver_port_element"/>
  </TemplatedPort>
  <ConnectorDefinition name="simple_message_connector"
                       pattern="message_interaction_pattern">
    <ConnectorPortDefinition api="message_emitter_port" name="emitter"
                             portRole="message_interaction_pattern.emitter"/>
    <ConnectorPortDefinition api="message_receiver_port" name="receiver"
                             portRole="message_interaction_pattern.receiver"/>
    <ItemBinding connectorItem="message_interaction_api::message_type_t"
                 interactionItem="message_interaction_pattern.message_item"/>
  </ConnectorDefinition>
</InteractionModule>
```

10.2 Standard properties (Normative, not mandatory)

This section defines standard properties. These properties can be associated with components to provide documentation.

```
<ContractModule name="standard_properties">
  <StringType size=”-1” baseChar=”CHAR8” name="property_string_t"/>
  <PropertyDefinition name="component_description_prop">
    <ConfigurationParameter name="description" type="property_string_t"/>
    <ConfigurationParameter name="category" type="property_string_t"/>
    <ConfigurationParameter name="version" type="property_string_t"/>
    <ConfigurationParameter name="vendor" type="property_string_t"/>
  </PropertyDefinition>
</ContractModule>
```

10.3 Advanced timer service (Normative, not mandatory)

The component execution policies defined in the core platform specifications (section 10.1.3) allow the definition of self-executing components: the business code of these components must implement a method run() that is called by the container. Though this minimalistic approach is convenient for nearly-hard real time applications, it may not be sufficient for more flexible cases, when the user code needs to reprogram timers. This section details the specification of user-programmable timers.

Two kinds of timers are defined: object-based and index-based.

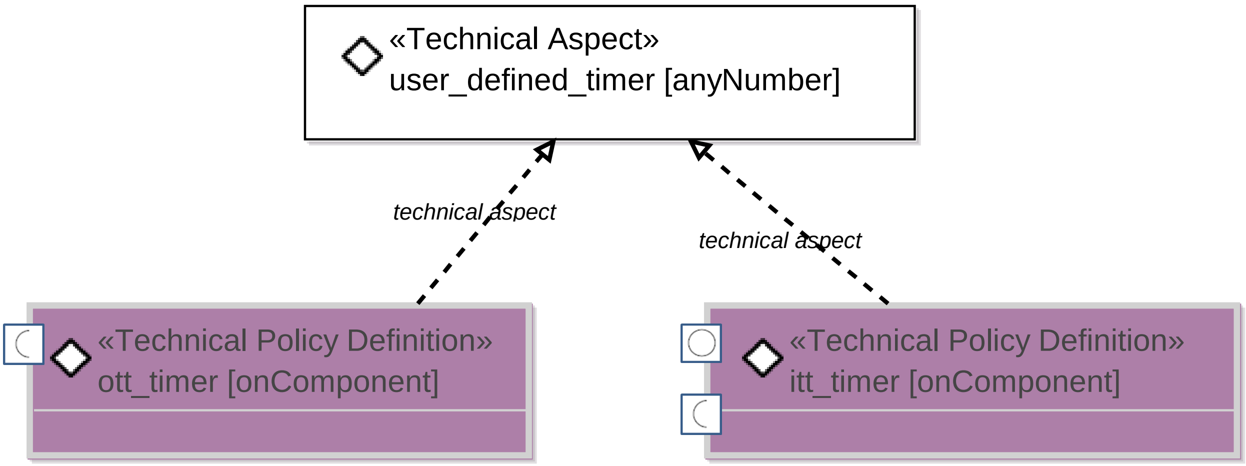

Figure 37: Standard timer policies

The declaration of the technical aspect for user defined timers is the following.

```
<NonfunctionalAspectModule name="timer">
  <NonfunctionalAspectModule name="itt_timer">
    […]
    <TechnicalPolicyDefinition applicability="onComponent" name="itt_timer"
                               technicalAspect="::timer::user_defined_timer">
      <PortElement interface="ucm_itt::itt_callback_intf" kind="PROVIDED"
                   name="timer_callback"/>
      <PortElement interface="ucm_itt::itt_service_intf" kind="REQUIRED"
                   name="timer_service"/>
    </TechnicalPolicyDefinition>
  </NonfunctionalAspectModule>
  <NonfunctionalAspectModule name="ott_timer">
    […]
    <TechnicalPolicyDefinition applicability="onComponent" name="ott_timer"
                               technicalAspect="::timer::user_defined_timer">
      <PortElement interface="ucm_ott::ott_scheduler" kind="REQUIRED"
                   name="timer_scheduler"/>
    </TechnicalPolicyDefinition>
  </NonfunctionalAspectModule>
  <TechnicalAspect multiplicity="anyNumber" name="user_defined_timer"/>
</NonfunctionalAspectModule>
```

Technical aspect “user_defined_timer” has cardinality “anyNumber”, meaning that an arbitrary number of user timer policies can be associated with a given component.

10.3.1 Object-based timers

The object-based timer policy implements a scheduler service that can deliver timer objects.

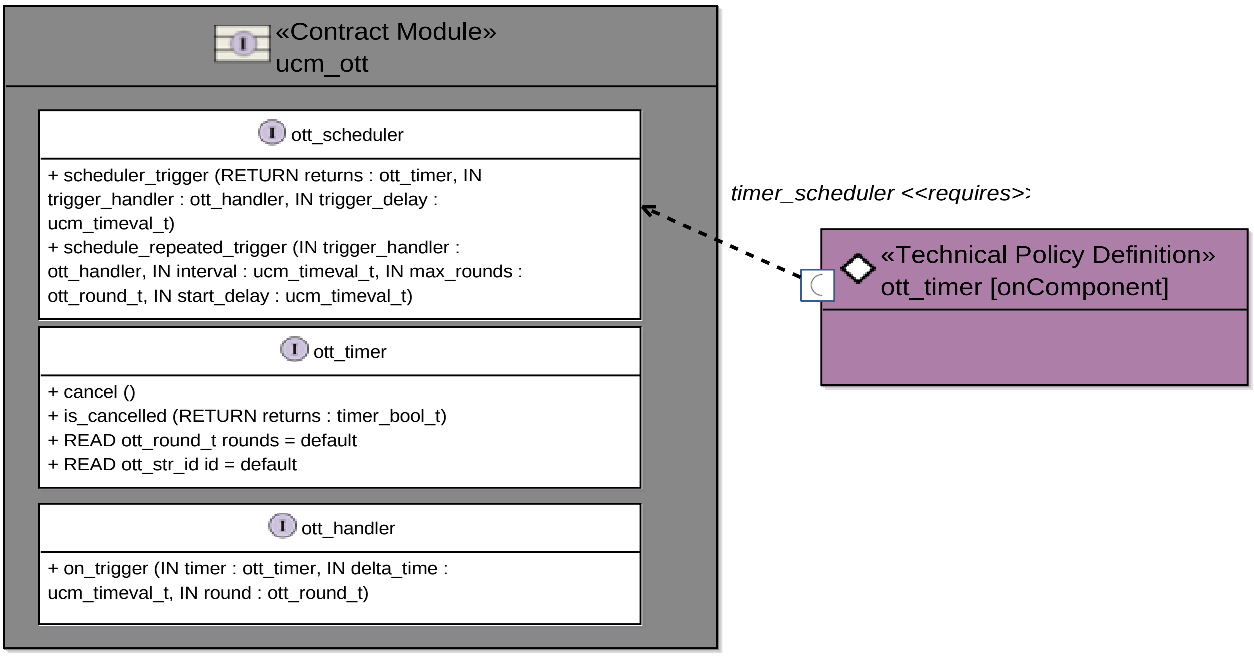

Figure 38: Technical policy for object-based programmable timers

The definition of the technical policy and associated contracts is illustrated on figure 38 and specified by the following XML declarations.

```
<NonfunctionalAspectModule name="ott_timer">
  <TechnicalPolicyDefinition applicability="onComponent" name="ott_timer"
                             technicalAspect="::timer::user_defined_timer">
    <PortElement interface="ucm_ott::ott_scheduler" kind="REQUIRED"
                 name="timer_scheduler"/>
  </TechnicalPolicyDefinition>
  <ContractModule name="ucm_ott">
    <!--UCM object-oriented timed trigger constract-->
    <PrimitiveInteger kind="ULONG" name="ott_round_t"/>
    <StringType charBase="CHAR8" name="ott_str_id"/>
    <Interface name="ott_handler">
      <Method name="on_trigger">
        <Parameter direction="IN" name="timer" type="ott_timer"/>
        <Parameter direction="IN" name="delta_time"
   type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
        <Parameter direction="IN" name="round" type="ott_round_t"/>
      </Method>
    </Interface>
    <Interface name="ott_timer">
      <Attribute defaultValue="" mode="READ" name="rounds" type="ott_round_t"/>
      <Attribute defaultValue="" mode="READ" name="id" type="ott_str_id"/>
      <Method name="cancel"/>
      <Method name="is_cancelled">
        <Parameter direction="RETURN" name="returns"
                   type="::timer::itt_timer::ucm_itt::timer_bool_t"/>
      </Method>
    </Interface>
    <Interface name="ott_scheduler">
      <Method name="scheduler_trigger">
```

```
        <Parameter direction="RETURN" name="returns" type="ott_timer"/>
        <Parameter direction="IN" name="trigger_handler" type="ott_handler"/>
        <Parameter direction="IN" name="trigger_delay"

type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
      </Method>
      <Method name="schedule_repeated_trigger">
        <Parameter direction="IN" name="trigger_handler" type="ott_handler"/>
        <Parameter direction="IN" name="start_delay"

type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
        <Parameter direction="IN" name="interval"

type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
        <Parameter direction="IN" name="max_rounds" type="ott_round_t"/>
      </Method>
    </Interface>
  </ContractModule>
</NonfunctionalAspectModule>
```

10.3.2 Index-based timers

Some real-time applications avoid relying on object-oriented concepts. For these applications, a simpler timer mechanism is defined.

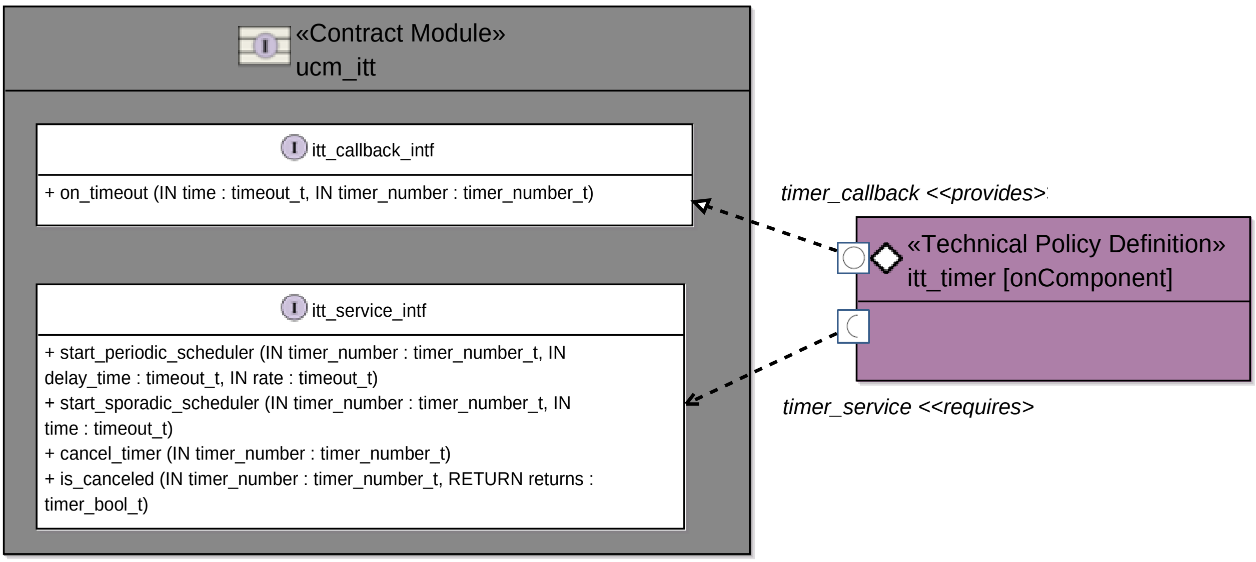

Figure 39: Technical policy for index-based programmable timers

The definition of the technical policy and associated contracts is illustrated on figure 39 and specified by the following XML declarations.

```
<NonfunctionalAspectModule name="itt_timer">
  <TechnicalPolicyDefinition applicability="onComponent" name="itt_timer"
                             technicalAspect="::timer::user_defined_timer">
    <PortElement interface="ucm_itt::itt_callback_intf" kind="PROVIDED"
                 name="timer_callback"/>
    <PortElement interface="ucm_itt::itt_service_intf" kind="REQUIRED"
                 name="timer_service"/>
  </TechnicalPolicyDefinition>
  <ContractModule name="ucm_itt">
    <!--UCM id-based timed trigger contract-->
    <Structure name="timeout_t">
```

```
      <StructureField name="time_val"
       type="::core::container_services::clock_service::clock_api::ucm_time_t"/>
      <StructureField name="flag" type="timeout_enum_t"/>
    </Structure>
    <Enumeration indexType="BYTE" name="timeout_enum_t">
      <Enumerator name="ABSOLUTE_TIME" value="0"/>
      <Enumerator name="RELATIVE_TIME" value="1"/>
    </Enumeration>
    <PrimitiveInteger kind="ULONG" name="timer_number_t"/>
    <PrimitiveBoolean name="timer_bool_t"/>
    <PrimitiveInteger kind="BYTE" name="timer_enum_index_t"/>
    <Interface name="itt_callback_intf">
      <Method name="on_timeout">
        <Parameter direction="IN" name="time" type="timeout_t"/>
        <Parameter direction="IN" name="timer_number" type="timer_number_t"/>
      </Method>
    </Interface>
    <Interface name="itt_service_intf">
      <Method name="start_periodic_scheduler">
        <Parameter direction="IN" name="timer_number" type="timer_number_t"/>
        <Parameter direction="IN" name="delay_time" type="timeout_t"/>
        <Parameter direction="IN" name="rate" type="timeout_t"/>
      </Method>
      <Method name="start_sporadic_scheduler">
        <Parameter direction="IN" name="timer_number" type="timer_number_t"/>
        <Parameter direction="IN" name="time" type="timeout_t"/>
      </Method>
      <Method name="cancel_timer">
        <Parameter direction="IN" name="timer_number" type="timer_number_t"/>
      </Method>
      <Method name="is_canceled">
        <Parameter direction="IN" name="timer_number" type="timer_number_t"/>
        <Parameter direction="RETURN" name="returns" type="timer_bool_t"/>
      </Method>
    </Interface>
  </ContractModule>
</NonfunctionalAspectModule>
```

Technical policy “itt_timer” has two port elements: one (“timer_callback”) is provided by the component executor, and must be implemented by the business code. It has a unique method “on_timeout”, which will be invoked upon timer expiration. The other port element (“timer_service”) is provided by the component context, and thus implemented by the component container. It has several methods to initiate a timer. A periodic timer will repeat infinitely; a sporadic timer will trigger once. Upon the initiation of a timer, the business code must provide a timer number. Thus, a single timer service can manage several timers, all being associated with the same callback method.

Timers can be canceled.

10.4 Additional interactions (Normative, not mandatory)

The core specifications defines APIs for service and message interactions (sections 10.1.5 and 10.1.6). This section defines additional interactions that are common in architectures Request-response is actually a bidirectional message-based interaction; it can easily be used for asynchronous communications. Shared data is a one-way data transmission in which receivers are notified and have to fetch updated versions of data–allowing to ignore some.

10.4.1 Request-response

The request-response interaction is a two-way communication. It is defined in a module named request-response.

Figure 40 illustrates the definition of the request-response interaction.

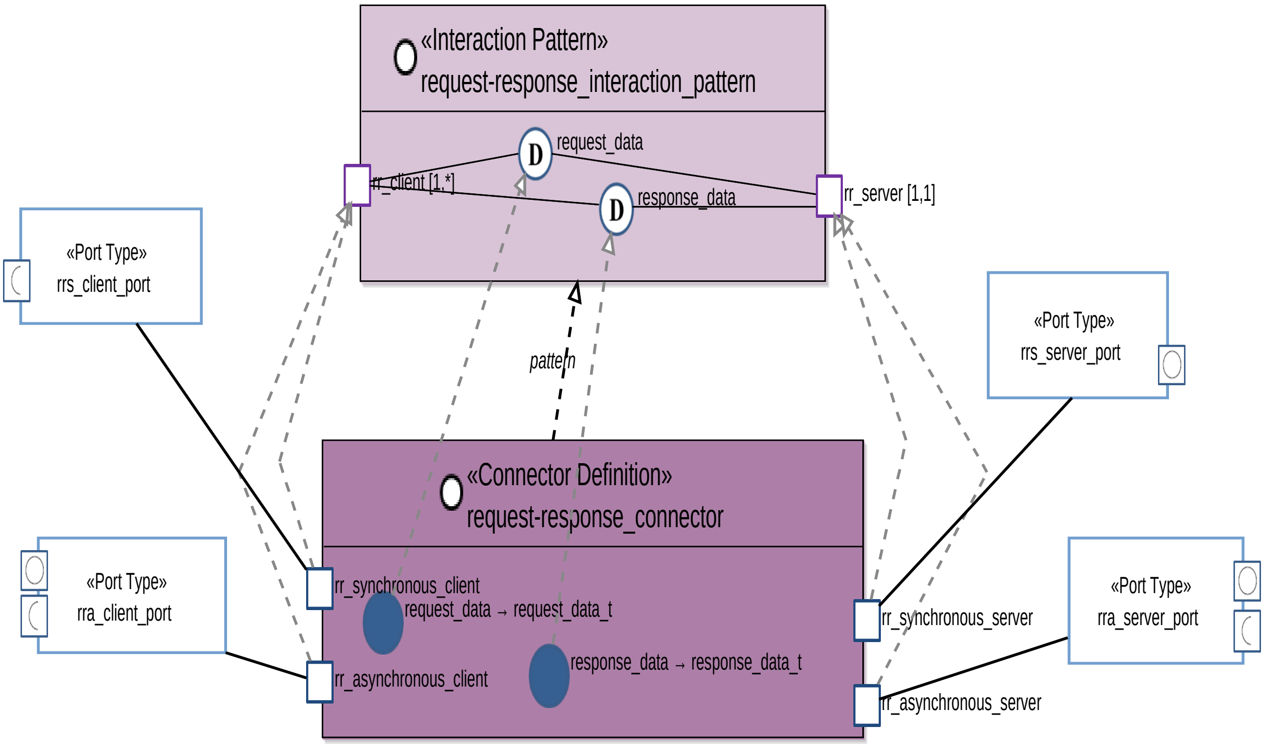

Figure 40: Request-response

The port type definitions are represented on figure 41.

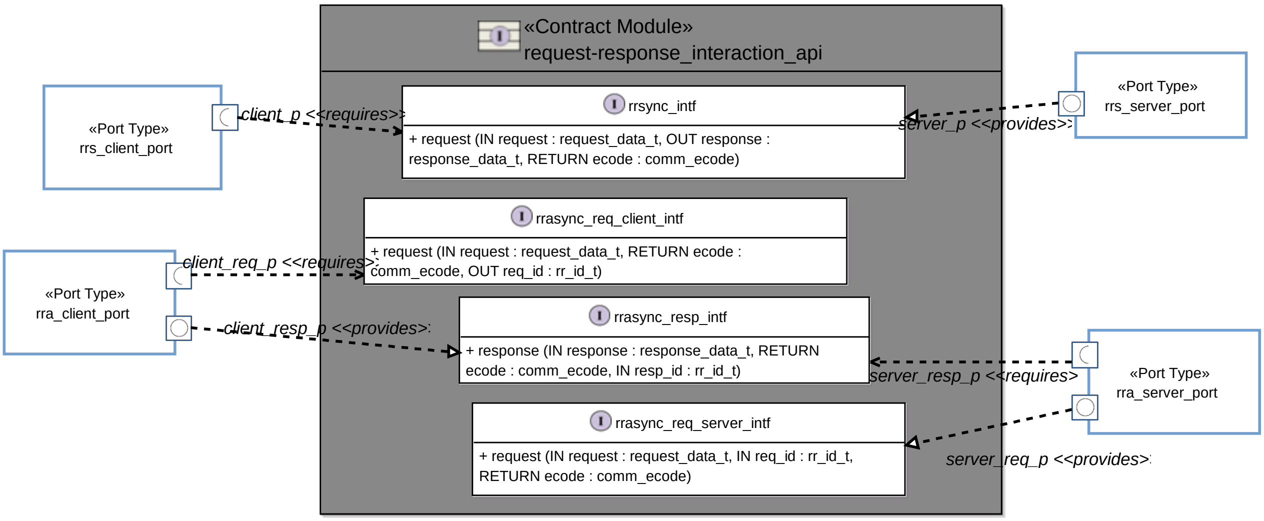

Figure 41: Port types for request-response interaction

10.4.1.1 Specifications

```
<InteractionModule name="request-response">
…
</InteractionModule>
```

It involves two interaction items: the request data and the response data. Two roles are defined: client and server. A request-response interaction involves a unique server, and at least one client.

```
<InteractionPattern name="request-response_interaction_pattern">
  <InteractionRole lowerMultipliciy="1" name="rr_client" upperMultipliciy="-1">
    <InvolvedItem item="request_data"/>
    <InvolvedItem item="response_data"/>
  </InteractionRole>
  <InteractionRole lowerMultipliciy="1" name="rr_server" upperMultipliciy="1">
    <InvolvedItem item="request_data"/>
    <InvolvedItem item="response_data"/>
  </InteractionRole>
  <InteractionItem name="request_data" nature="DATA"/>
  <InteractionItem name="response_data" nature="DATA"/>
</InteractionPattern>
```

Several APIs are defined: an interface rrsync_intf for synchronous communications (on client and server side), and a couple of interfaces (rrasync_req_intf and rrasync_resp_intf) for asynchronous communications (on client and server side). The interfaces for asynchronous communications allow for decoupling the reception of the request data and the emission of the response data.

```
<ContractModule name="request-response_interaction_api">
  <DataTypeTemplateParameter name="request_data_t"/>
  <DataTypeTemplateParameter name="response_data_t"/>
  <PrimitiveType kind="ULONG" name="rr_id_t"/>
  <Interface name="rrsync_intf">
    <!--interface for request-response synchronous client and server-->
    <Method name="request">
      <Parameter direction="IN" name="request" type="request_data_t"/>
      <Parameter direction="OUT" name="response" type="response_data_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
  </Interface>
  <Interface name="rrasync_req_server_intf">
    <!--interface for request-response asynchronous server (request)-->
    <Method name="request">
      <Parameter direction="IN" name="request" type="request_data_t"/>
      <Parameter direction="IN" name="req_id" type="rr_id_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
  </Interface>
  <Interface name="rrasync_resp_intf">
    <!--interface for request-response asynchronous client and server
(response)-->
    <Method name="response">
      <Parameter direction="IN" name="response" type="response_data_t"/>
      <Parameter direction="IN" name="resp_id" type="rr_id_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
  </Interface>
  <Interface name="rrasync_req_client_intf">
    <!--interface for request-response asynchronous client (request)-->
    <Method name="request">
      <Parameter direction="IN" name="request" type="request_data_t"/>
      <Parameter direction="OUT" name="req_id" type="rr_id_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
```

```
    </Method>
  </Interface>
</ContractModule>
```

A set of template ports carry these interfaces to define the different possible connector port specifications.

```
<TemplatedPort name="rrs_client_port">
  <PortElement interface="request-response_interaction_api::rrsync_intf"
               kind="REQUIRED" name="client_p"/>
</TemplatedPort>
<TemplatedPort name="rrs_server_port">
  <PortElement interface="request-response_interaction_api::rrsync_intf"
               kind="PROVIDED" name="server_p"/>
</TemplatedPort>
<TemplatedPort name="rra_client_port">
  <PortElement interface="request-
response_interaction_api::rrasync_req_client_intf"
               kind="REQUIRED" name="client_req_p"/>
  <PortElement interface="request-response_interaction_api::rrasync_resp_intf"
               kind="PROVIDED" name="client_resp_p"/>
</TemplatedPort>
<TemplatedPort name="rra_server_port">
  <PortElement
          interface="request-response_interaction_api::rrasync_req_server_intf"
          kind="PROVIDED" name="server_req_p"/>
  <PortElement interface="request-response_interaction_api::rrasync_resp_intf"
               kind="REQUIRED" name="server_resp_p"/>
</TemplatedPort>
```

The connector definition itself defines four possible ports: two for the client role (synchronous and asynchronous), and two for the server role (synchronous and asynchronous). As the interaction pattern specifies there can only be a unique server, either the synchronous server port or the asynchronous server port shall be connected.

```
<ConnectorDefinition name="request-response_connector"
                     pattern="request-response_interaction_pattern">
  <ConnectorPortDefinition api="rrs_client_port" name="rr_synchronous_client"
                    portRole="request-response_interaction_pattern.rr_client"/>
  <ConnectorPortDefinition api="rrs_server_port" name="rr_synchronous_server"
                    portRole="request-response_interaction_pattern.rr_server"/>
  <ConnectorPortDefinition api="rra_client_port" name="rr_asynchronous_client"
                    portRole="request-response_interaction_pattern.rr_client"/>
  <ConnectorPortDefinition api="rra_server_port" name="rr_asynchronous_server"
                    portRole="request-response_interaction_pattern.rr_server"/>
  <ItemBinding connectorItem="request-response_interaction_api::request_data_t"
          interactionItem="request-response_interaction_pattern.request_data"/>
  <ItemBinding connectorItem="request-response_interaction_api::response_data_t"
          interactionItem="request-response_interaction_pattern.response_data"/>
</ConnectorDefinition>
```

10.4.1.2 Semantics

Synchronous client and server ports have the same execution semantics as in the service connector (§ 10.1.5): clients send the request data to the server and await the reception of the response data.

Asynchronous ports allow deferred computation. The processing of the response data is performed by a callback in asynchronous clients. On server side, incoming request data can be stored to be processed later; the response API can be invoked anytime. The identifier parameters req_id and resp_id are used to ensure the correspondence between the request and the response. It is thus possible for a client to send several requests before processing the responses. The same way, a server can receive several requests before sending responses.

10.4.2 Shared data

The shared data interaction is meant to be used for data transmission between several writers and several readers. Unlike the message interaction (section 10.1.6), readers fetch data whenever they need to, instead of receiving messages. On the writer side, data is written and sent (or canceled) using two different methods, thus allowing to set data values and publish them at different paces. Figure 42 illustrates the definition of the shared data interaction.

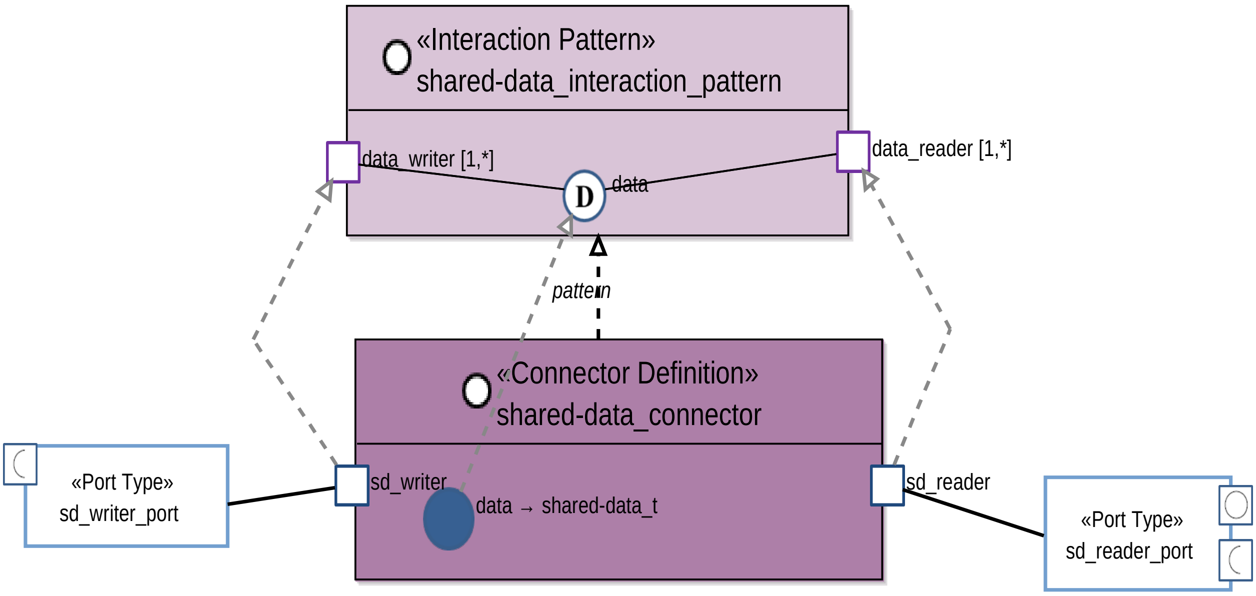

Figure 42: Shared data

Figure 43 illustrates the definitions of the port types involved in shared data interactions.

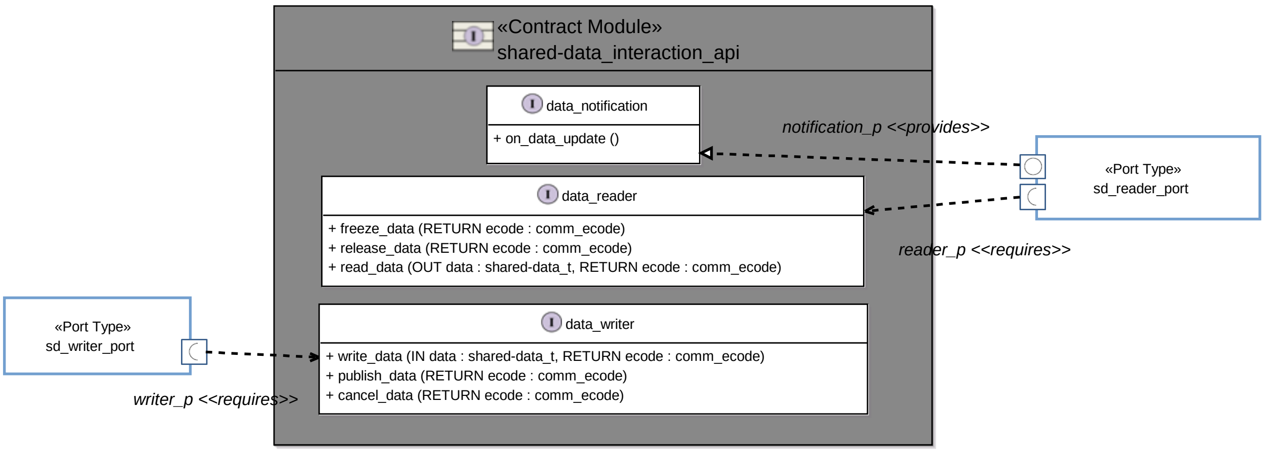

Figure 43: Port types for shared data interactions

10.4.2.1 Specifications

The shared data interaction is defined in an interaction module

```
<InteractionModule name="shared-data">
…
</InteractionModule>
```

Three interfaces are defined: one for the publication, one for update notification, and one for reception. They manipulate a template data parameter named “shared-data_t”, which represents the actual shared data.

```
<ContractModule name="shared-data_interaction_api">
  <DataTypeTemplateParameter name="shared-data_t"/>
  <Interface name="data_reader">
    <Method name="freeze_data">
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
    <Method name="release_data">
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
    <Method name="read_data">
      <Parameter direction="OUT" name="data" type="shared-data_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
  </Interface>
  <Interface name="data_notification">
    <Method name="on_data_update">
      <!--no error code for this method, since it is called by the connector-->
    </Method>
  </Interface>
  <Interface name="data_writer">
    <Method name="write_data">
      <Parameter direction="IN" name="data" type="shared-data_t"/>
      <Parameter direction="RETURN" name="ecode"
                 type="::core::return_codes::comm_ecode"/>
    </Method>
    <Method name="publish_data">
       <Parameter direction="RETURN" name="ecode"
                  type="::core::return_codes::comm_ecode"/>
    </Method>
    <Method name="cancel_data">
       <Parameter direction="RETURN" name="ecode"
                  type="::core::return_codes::comm_ecode"/>
    </Method>
  </Interface>
</ContractModule>
```

Finally, the interaction pattern and the connector are defined.

```
<InteractionPattern name="shared-data_interaction_pattern">
  <InteractionRole lowerMultipliciy="1" name="data_writer"
                   upperMultipliciy="-1">
    <InvolvedItem item="data"/>
  </InteractionRole>
  <InteractionRole lowerMultipliciy="1" name="data_reader"
                   upperMultipliciy="-1">
    <InvolvedItem item="data"/>
  </InteractionRole>
  <InteractionItem name="data" nature="DATA"/>
</InteractionPattern>
<TemplatedPort name="sd_writer_port">
  <PortElement interface="shared-data_interaction_api::data_writer"
               kind="REQUIRED"
               name="writer_p"/>
</TemplatedPort>
<TemplatedPort name="sd_reader_port">
  <PortElement interface="shared-data_interaction_api::data_reader"
               kind="REQUIRED"
               name="reader_p"/>
  <PortElement interface="shared-data_interaction_api::data_notification"
```

```
                kind="PROVIDED"
                name="notification_p"/>
</TemplatedPort>
<ConnectorDefinition name="shared-data_connector"
                     pattern="shared-data_interaction_pattern">
  <ConnectorPortDefinition api="sd_reader_port" name="sd_reader"
                       portRole="shared-data_interaction_pattern.data_reader"/>
  <ConnectorPortDefinition api="sd_writer_port" name="sd_writer"
                       portRole="shared-data_interaction_pattern.data_writer"/>
  <ItemBinding connectorItem="shared-data_interaction_api::shared-data_t"
                interactionItem="shared-data_interaction_pattern.data"/>
</ConnectorDefinition>
```

10.4.2.2 Semantics

The reader API has two port elements: reader_p to fetch data, and notification_p to be notified of data updates. The notification port is called by the connector upon data update. The reader port has three methods: freeze_data(), release_data() and read_data().

Method read_data gets the current value of the shared data. Method freeze_data prevents the data value from being updated, thus allowing the reader to work on a stable value. Method release_data is the opposite of freeze_data: it allows the updates of the data value.

The writer API has one port element, which is provided by the connector. This port element has three methods: write_data(), publish_data() and cancel_data().

Method write_data() sets a value for the shared data, but does not send it. Method publish_data() actually sends the data value set by write_data(). Method cancel_data() voids the value set by write_data(). Consequently, calling publish_data_() after cancel_data() shall have no effect.

10.5 Additional component execution policies (Normative, not mandatory)

This section describes extensions to the “protected self-executing component” and “protected active component” technical policy (§ 10.1.3). Additional technical policies are defined to specify more detailed execution semantics: self-executing components with periodic or one-shot, background execution; active components with periodic or sporadic execution.

10.5.1 Specifications

The two technical policies periodic_self-executing_component and background_self-executing_component extend technical policy self-executing_component. They add configuration parameters to specify task priority, etc. Figure 44 illustrates these definitions.

Task priority is used for scheduler configuration and scheduling analysis. It is a value between 0 and 255 (unsigned byte), 1 corresponding to the highest priority, and 255 being the lowest priority. Offset corresponds to the delay between the start of the system and the actual start of the task.

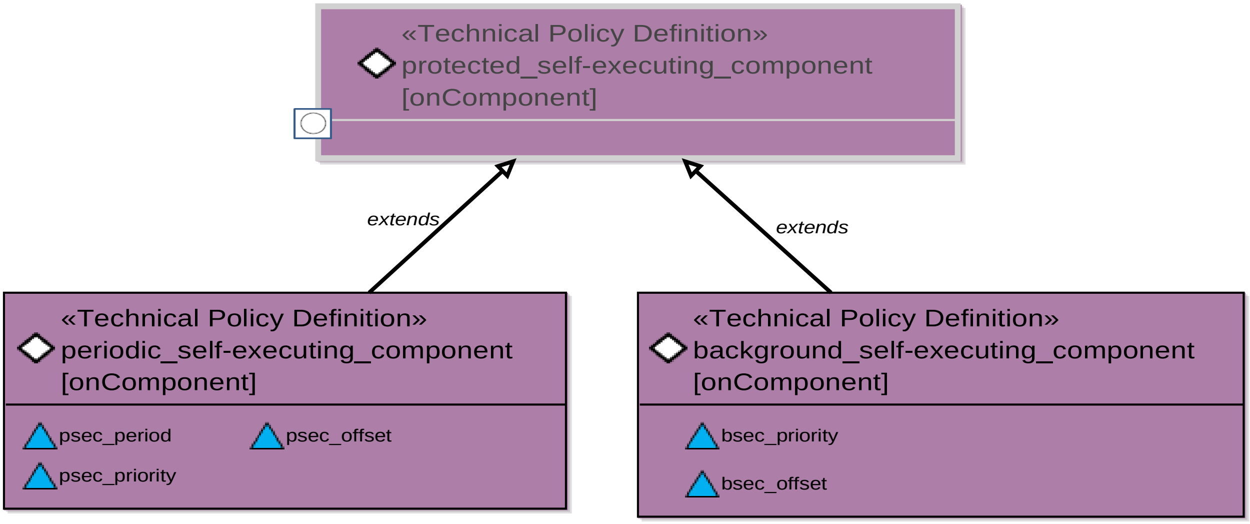

Figure 44: Extended technical policies for self-executing components

The two technical policies periodic_protected_active_component and sporadic_protected_active_component extend protected_active_component. They add configuration parameters to specify task priority, etc. Figure 45 illustrates these definitions.

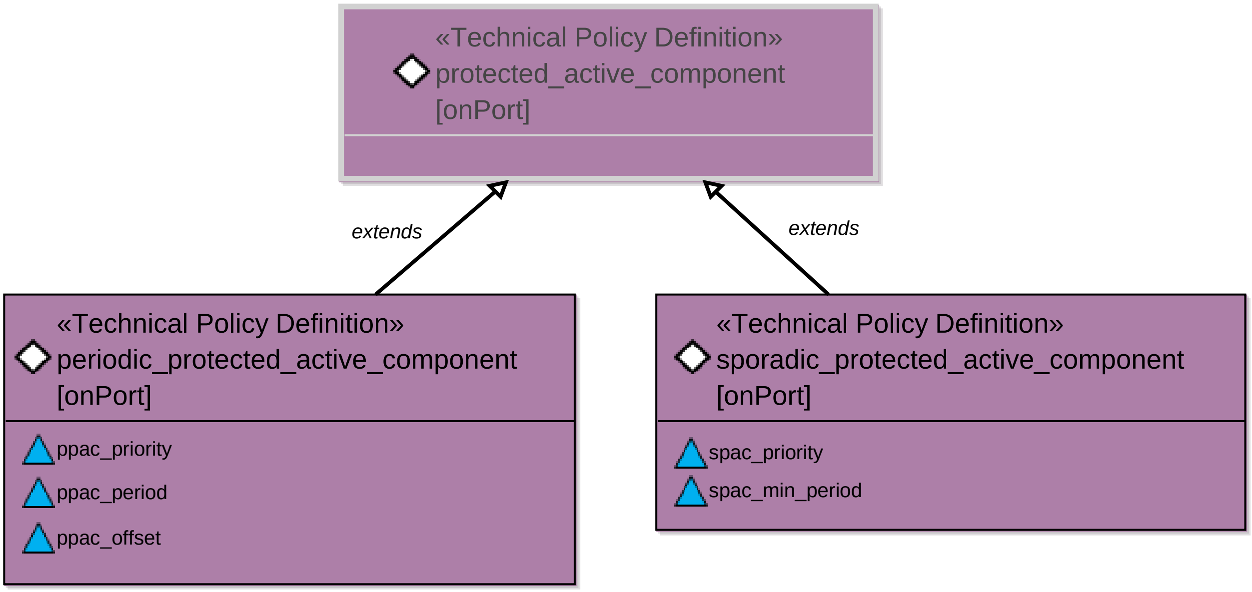

Figure 45: Extended technical policies for active components

The following XML declarations correspond to the definition of the four technical policies.

```
<NonfunctionalAspectModule name="execution_policies">
  <TechnicalPolicyDefinition applicability="onComponent"
   extends="::component_execution_policies::protected_self-executing_component"
   name="periodic_self-executing_component"
   technicalAspect="::component_execution_policies::component_execution_policy">
    <ConfigurationParameter name="psec_period"
    type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
    <ConfigurationParameter name="psec_priority"
                            type="execution_contracts::priority_t"/>
    <ConfigurationParameter name="psec_offset"
```

```
    type="core::container_services::clock_service::clock_api::ucm_timeval_t"/>
  </TechnicalPolicyDefinition>
  <TechnicalPolicyDefinition applicability="onComponent"
   extends="::component_execution_policies::protected_self-executing_component"
   name="background_self-executing_component"
   technicalAspect="::component_execution_policies::component_execution_policy">
    <ConfigurationParameter name="bsec_priority"
                            type="execution_contracts::priority_t"/>
    <ConfigurationParameter name="bsec_offset"
    type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
  </TechnicalPolicyDefinition>
  <TechnicalPolicyDefinition applicability="onPort"
   extends="::component_execution_policies::protected_active_component"
   name="sporadic_protected_active_component"
   technicalAspect=
       "::core::component_execution_policies::component_execution_policy">
    <ConfigurationParameter name="spac_priority"
                            type="execution_contracts::priority_t"/>
    <ConfigurationParameter name="spac_min_period"
    type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
  </TechnicalPolicyDefinition>
  <TechnicalPolicyDefinition applicability="onPort"
   extends="::core::component_execution_policies::protected_active_component"
   name="periodic_protected_active_component"
   technicalAspect=
       "::core::component_execution_policies::component_execution_policy">
    <ConfigurationParameter name="ppac_priority"
                            type="execution_contracts::priority_t"/>
    <ConfigurationParameter name="ppac_period"
    type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
    <ConfigurationParameter name="ppac_offset"
    type="::core::container_services::clock_service::clock_api::ucm_timeval_t"/>
  </TechnicalPolicyDefinition>
  <ContractModule name="execution_contracts">
    <PrimitiveInteger kind="UBYTE" name="priority_t">
      <!--priority 1 is the highest-->
    </PrimitiveInteger>
  </ContractModule>
</NonfunctionalAspectModule>
```

10.5.2 Semantics

For periodic and background self-executing components, method run is called according to the configuration parameters. Every period, with an offset for the periodic execution. Once, after the offset for the background execution.

For periodic and sporadic active components, the execution is triggered upon port invocation. For periodic Every period, with an offset for the periodic execution. Whenever an invocation occurs, with a minimum delay between two executions for the sporadic execution.

The extended technical policies make no assumptions regarding the underlying execution threads that would support the executions. This depends on the actual implementation choices made by the platform provider. Priorities are used for scheduling computation.

11. UCM Programming Model

This section describes the standard way component implementations are structured, and the corresponding API. The main element is the container, which contains all the runtime elements of a component.

The container is the component's implementation runtime environment. It is a framework that integrates a set of technical policies and connectors implementations with the component's behavior. It allows the component's implementation to benefit from both the technical policies and the connectors support. The technical policies implementations manage the technical aspects on behalf of the components. The connectors implementations ensure inter-components interactions.

In order to enforce the UCM container extensibility, its capabilities are designed following a component-based approach. Thus, the connectors and the technical policies implementations are themselves comparable to a set of components implementations. Their interactions with the user business code of the component require explicit connections between their port elements. This means that all the dependencies between the connectors, the technical policies and the business logic inside a given component are clearly expressed by the ports (for connectors) or features (for technical policies), whatever the dependency is on the infrastructure or on other application components. This approach allows to leverage the components portability and reuse as all their dependencies are captured and managed by their containers.

11.1 Runtime entities

11.1.1 Component implementation: Component Body

The component body is the programmatic element that maps to the *AtomicComponentImplementation* element as defined by the UCM PIM. It supplies the component business logic only. It concentrates on realizing the component behavior without caring of any non functional aspect. The component body is hosted by a container that manages its life cycle and complement it by the technical support that allows it to run. Concretely, the component body is a set of programming language-specific artifacts that are defined by the different language mappings that are specified by the UCM specification.

11.1.2 Connector and technical policies implementation: Fragments

The non functional support in a UCM runtime is provided by the technical policies and the connectors implementations elements. They are designed as a set of components called “Fragments”. A fragment is similar to a functional component body. This is because a connector implementation, as a component, owns a set of configuration attributes and port elements. Similarly, a technical policy definition also owns a set of configuration attributes and port elements. So, at the programming level, the components, the connectors and the technical policies may be managed in the same way. This approach allows to modularize the UCM runtime as most as possible to ease its extensibility

A fragment is deployed in the same way as a user component. It is also hosted by a container that manages its life cycle. If needed, the interactions between the components and the fragments are performed using explicit connections between their ports elements. The difference between a fragment and a user component implementation is in its interactions with its container. The fragment may need to collaborate with the container to perform its functionality. It has a special access to the container interfaces. Although appreciated, the UCM specification does not target fragments portability over different UCM frameworks. The complexity of some non functional behaviors may require a strong adherence of the fragment implementation to the underlying framework. In fact, a technical policy may act in two ways. Either explicitly, as a service that directly invokes the functional component and/or is invoked by it using port elements; and/or implicitly, without any port element. In this last case, the fragment may need some extended capabilities that are out of scope of this specification.

A connector implementation, as well as a technical policy implementation, are realized by one or more fragments. The mapping of their definitions as defined by the UCM meta-model onto fragments is up to the platform provider. The following two subsections provide some hints to how to transform a UCM Connector (resp. Technical Policy) to a set of fragments.

11.1.2.1 From connectors to fragments (not normative)

As states by the UCM metamodel, a ConnectorDefinition owns a set of ConnectorPortDefinitions that include, similarly to a component Port type, a set of PortElements, knowing that a PortElement is an abstraction of a provided or required interface. A connector definition is concretely implemented by a set fragments. A fragment is necessarily co-localized to the component using it. Each fragment will realize a part of the interaction, by implementing one or more PortElements. The mapping of the connector PortElements to fragments is implementation-dependent. Figure 46 depicts an example of that mapping. In that example, each PortElement is realized by a separate fragment.

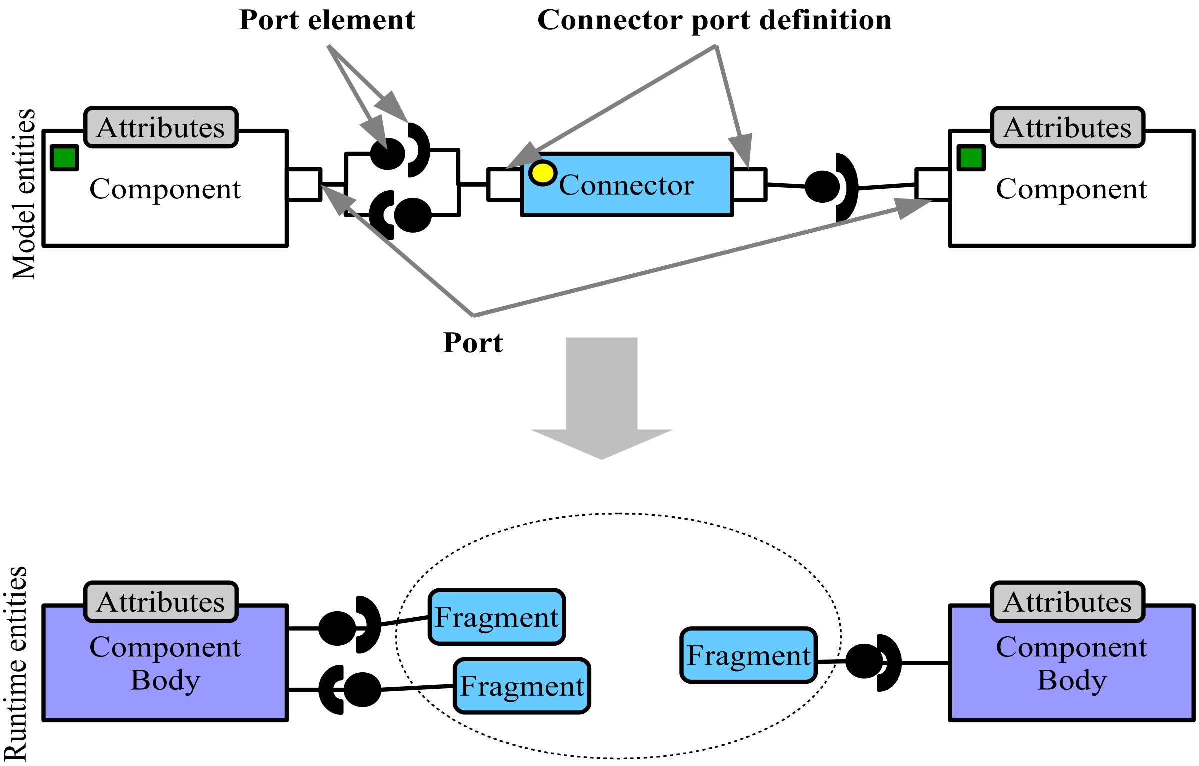

Figure 46: Connector fragmentation example

The communication between the fragments is connector-specific. It is typically based on the communication mechanisms that the connector is intended to abstract. Ex: the fragments of DDS-based connector implementation will typically interact via DDS (at least), the fragments of a shared memory-based connector will use that same mechanism to interact.

11.1.2.2 From technical policies to fragments (not normative)

At the model level, the AtomicComponentImplementation that implements a given ComponentDefinition is associated to a TechnicalPolicyDefinition. This latter owns also a set of PortElements. At runtime, these PortElements are implemented by one or more fragments. As for the connector PortElements, the mapping to fragments is implementation-dependent. Figure 47 shows an example where all the PortElements of the TechnicalPolicy are realized by one fragment.

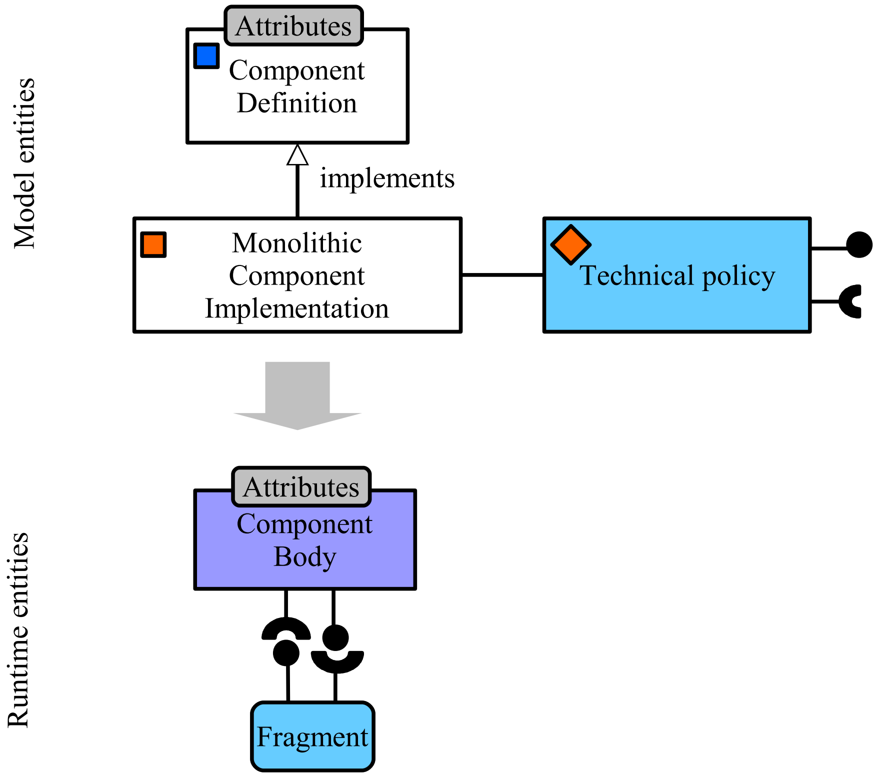

Figure 47: Technical policy fragmentation example

11.1.3 Container

The container is the glue that allows the component implementations to collaborate with the fragments to make them operational. The main container role is to manage the components life cycle and enable the communication between them. It also enables some additional technical policies by managing and collaborating with them as a set of fragments.

A container may wrap multiple component implementations and multiple fragments. Multiple containers could coexist within a UCM runtime instance. Basically, a Container defines a technical management scope that is common to all the belonging components. It typically defines a common life cycle management strategy applied to all the included components. Creating multiple Containers allows, for instance, to apply multiple life cycle and execution policies on different groups of components instances. It does not mean that different containers cannot apply the same life cycle strategy.

11.2 Container programming model

The container programming model defines the different standard interfaces between the different UCM runtime elements, including the container, the Component body and the fragments. Figure 48 shows the different interactions that could exist within a UCM runtime instance, and those that are specified by the UCM standard and those that are not. The main goal of the container programming model is to be able to implement portable component bodies. That's why all the interactions of the Component body with its environment must be clearly specified.

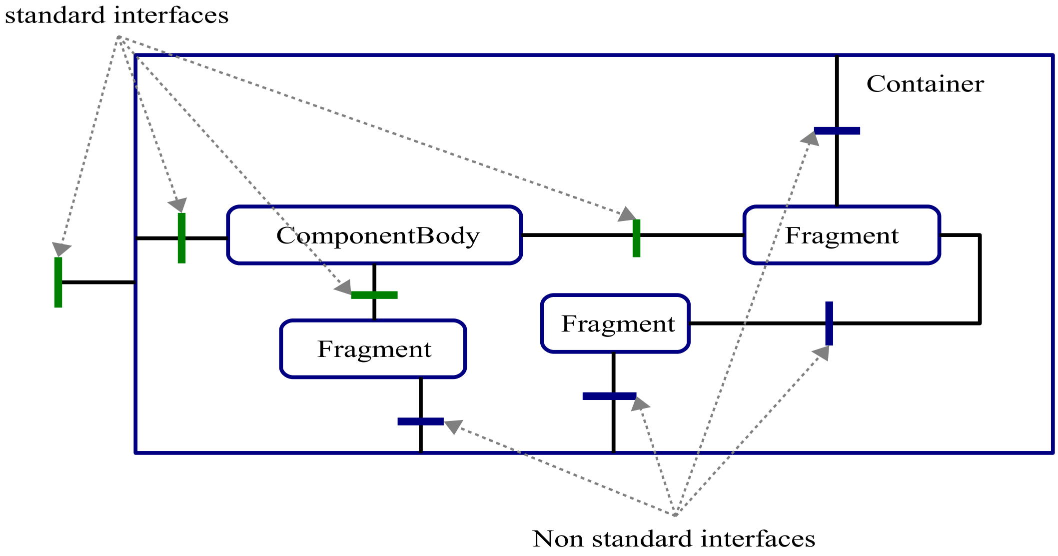

Figure 48: UCM Runtime Interfaces

Figure 49 depicts the UML model of the UCM container programming model elements. They are described in the following.

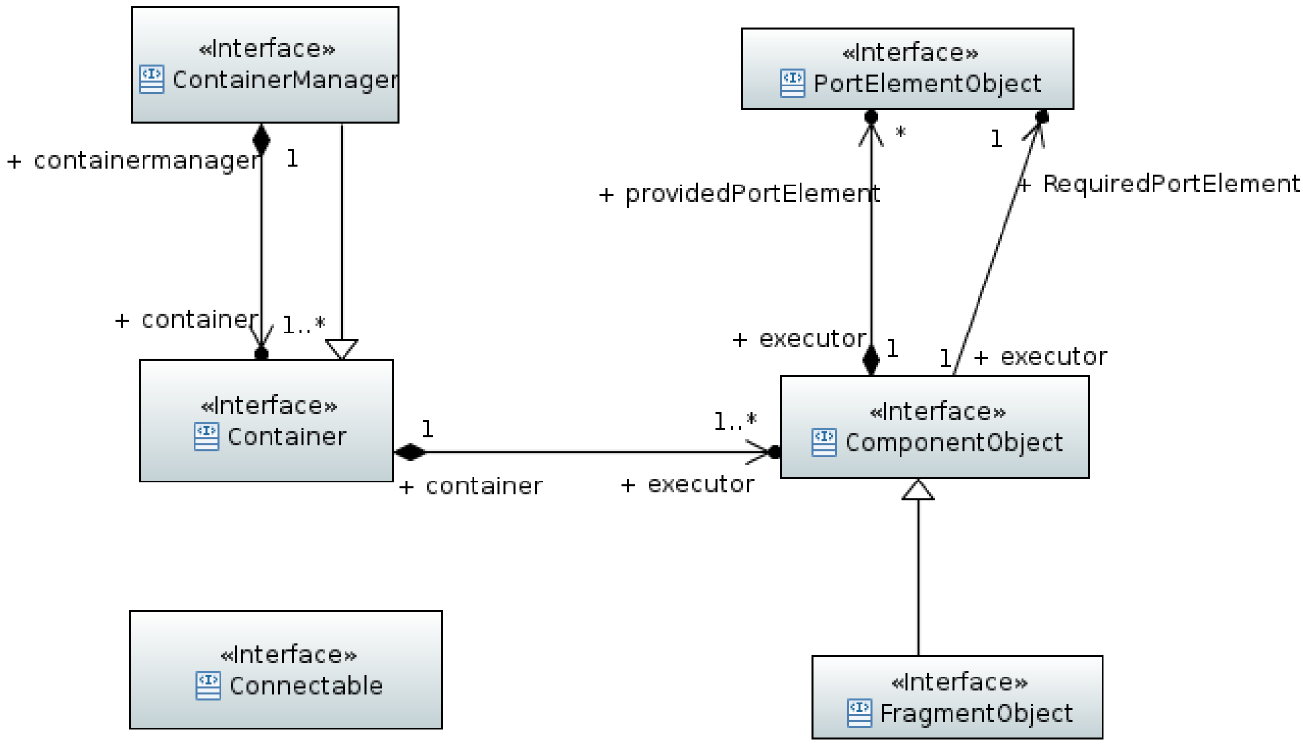

Figure 49: UCM Container Programming Model

A *Container* is defined as an aggregation of *ComponentObject* entities representing component bodies, and *FragmentObject* entities representing fragments. A *ComponentObject*, as a *FragmentObject*, includes a set of *PortElementObject* entities representing their provided port elements, and references others representing their required port elements. The following sections describe these entities.

11.2.1 Component interfaces

Figure 50 shows the different interfaces of a UCM component whatever it is a functional component or a fragment.

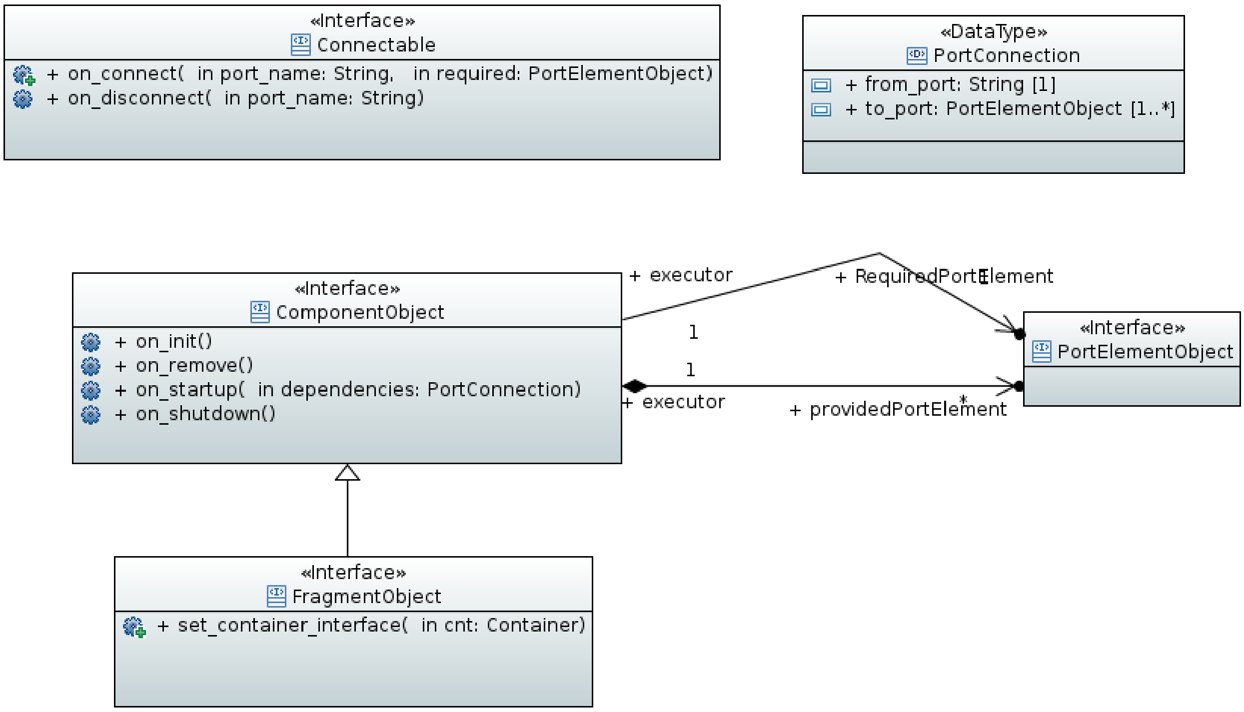

Figure 50: UCM Component Body Interfaces

11.2.1.1 PortElementObject

The *PortElementObject* characterizes any UCM port element interface, whatever it belongs to a component or a connector or a technical policy. All the UCM ports elements implementation shall support that interface. A *PortElementObject* shall have the methods specified in the interface associated with the port element (§ 9.3.5.2), after having applied the possible data type bindings (§ 9.5.3.7). It holds the business logic of that interface.

11.2.1.2 ComponentObject

The *ComponentObject* interface represents a UCM component body. It is the interface between the component body and its container. It allows the container to notify the component of its lifetime changes from its creation to its removal (§ 11.2.3) passing by its operational phase. A *ComponentObject* creates its provided port elements represented by the *PortElementObject* interface. The provided *PortElementObjects* hold the business logic of the provided ports elements. A *ComponentObject* references its required *PortElementObjects*, in other words its dependencies. These dependencies are resolved by the container and provided to the *ComponentObject* when it starts its operational phase.

The following items describes the *ComponentObject* methods:

Method **on_init()** is called by the container to allow the component to initialize its internal state prior to its startup. If the component exhibits a set of attributes whose initialization is driven by an external deployment tool, the *on_init* method should be called once the component attributes have been initialized.

Method **on_remove()** is called by the container to notify the component that it is about to be removed.

Method **on_startup()** is called by the container to allow the component to start its operational phase, where it is ready to interact with other components. This call signals the end of the whole application configuration, including its components initialization and connections. The container provides the component required ports references as a parameter of the on_startup method, so that the component can start using them. These references are presented to the component as a collection of ***PortConnection*** objects including the component provided port element name and its required *PortElementObject* reference.

Method **on_shutdown()** notifies the component of the end of its operational phase. The component should typically release any resources it acquired at startup time.

11.2.1.3 Connectable

The *Connectable* interface is a callback interface that the component body can optionally implement to be notified of each individual port element connection and disconnection. A component may be interested in those events to initialize some data that is related to these connections. As this interface is called while the component is still at its configuration phase, the component should not use its ports.

Method **on_connect()** notifies the component that its port element as named by the first parameter has been connected to the *PortElementObject* as referenced by the second parameter. Hence, the on_connect method will be called as many times as the component has required ports elements.

Method **on_disconnect()** notifies the component of the disconnection of the port element as named by this method parameter.

11.2.1.4 FragmentObject

The *FragmentObject* interface represents a UCM fragment, whatever it is a connector fragment or a technical policy fragment. As stated before, a fragment implementation is similar to a component body, that is why the *FragmentObject* interface extends the *ComponentObject* one. A *FragmentObject* lifetime is managed by its container in the same way as a *ComponentObject*. The only difference between the *FragmentObject* and the *ComponentObject* interfaces is the ability of the first one to access to its container interface in order to collaborate with it when needed. This is the intent of the set_container_interface method.

Method **set_container_interface()** is called by the container to provide the fragment of an access point to itself, so that it can get information about the belonging *ComponentObjects* and act on them if needed.

Note that the interface between the fragment and the container is not completely specified, as fragment portability is not aimed in this specification. It is considered that the minimum that a container could exhibit to its fragments is what it already exhibits to the deployment tool (the *Container* interface).

11.2.2 Container interfaces

Figure 51 depicts the Container-related interfaces.

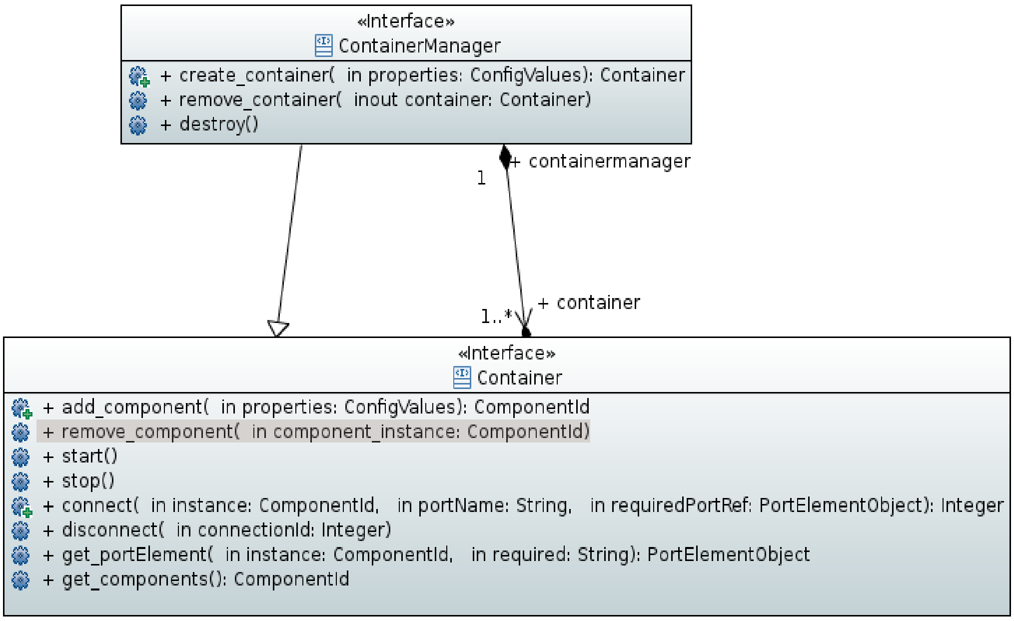

Figure 51: Container and container manager

11.2.2.1 Container

The *Container* interface exposes a management API that allows the deployment of a set of components and fragments. A container is able to instantiate arbitrary *ComponentObjects* and *FragmentObject* instances and manage their lifetime. It provides a set of methods that allows to instantiate, initialize, connect and start these entities. They are described in the following items.

Method **add_component()** allows to create and initialize a given UCM component body or fragment instance. A set of configuration values are passed as parameter to provide the information needed to create a *ComponentObject* or *FragmentObject* instance. *add_component* returns a unique identifier for the created entity.

Note that add_component() does not return a reference to *ComponentObject* or *FragmentObject* because these interfaces are internal interfaces. They define the interaction between the container and its components only. They do not have to be exhibited to third parties.

Method **remove_component()** allows to remove a given component identified by its identifier.

Method **get_portElement()** allows to get a provided *PortElementObject* reference of an existing *ComponentObject* instance. This reference is typically used for connecting it to a component port element that requires it.

Method **connect()** allows to connect a component port element, identified by its name, to a *PortElementObject* provided by another *ComponentObject*. It returns a connection identifier that will be used for undoing this connection.

Method **disconnect()** allows to disconnect a component-to-component connection previously established.

Method **start()** signals the completion of the configuration phase and the beginning of the operational phase for all the *ComponentObjects* of the current Container. Calling this method will start all the included *ComponentObjects*.

Method **stop()** signals the completion of the operational phase for all the included *ComponentObjects*. Calling this method will stop them in their startup-reverse order.

Method **get_component()** allows to get a list of all the included *ComponentObjects* identifiers.

11.2.2.2 ContainerManager

The *ContainerManager* interface characterizes the root container that represents a UCM runtime instance. It allows to create and remove component bodies, fragments and other containers. In addition to the *Container* base methods, this interface provides the following methods.

Method **create_container()** creates and configure a Container instance with the provided configuration values parameter.

Method **remove_container()** removes an existing Container instance. The removal of a container instance implies the shutdown and removal of its included entities.

Method **destroy()** terminates a UCM runtime instance and frees all associated resources by removing all the included containers, components and fragments.

11.2.3 Component life cycle management

Two main phases should be distinguished in a UCM component lifetime at runtime: configuration phase and operational phase.

In the configuration phase, a component instance is initialized and connected to its dependencies. A component instance is initialized by setting its attributes. Component attributes are intended to be used to tune the component behavior for a specific application use case. Once initialized, if the component has defined required ports element, these ports are connected to other compatible ports elements. During this phase, the component ports are disabled. It cannot either invoke other components, or be invoked by others.

In the operational phase, all the application components instances are ready to run and to collaborate together to achieve the application functional purpose. All the components interactions can start. Typically, this phase is where the component execution policy goes in action.

Distinguishing between these two phases guarantees that all the application components are set up before they start to run. It allows to avoid the errors that can happen if one component starts to interact with partially configured components. Serializing between the configuration and the operational phases is particularly required in highly connected component-based applications.

Figure 52 Shows the different states that the component passes through during these two phases.

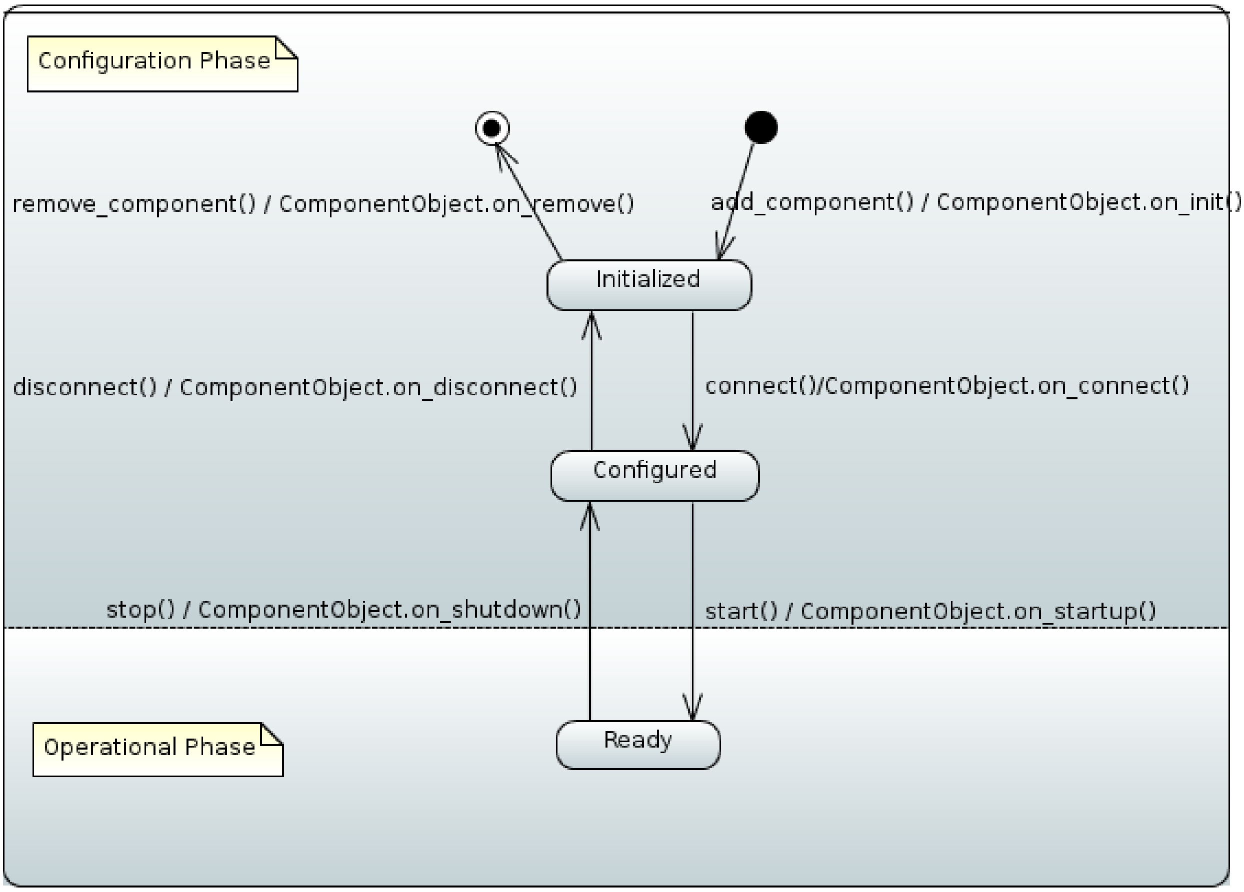

Figure 52: UCM Component Instance Life Cycle

As stated before, a UCM component instance life cycle is driven by its container as follows:

The component instance is ***initialized*** when the *add_component* method is called on the Container. The component body is then instantiated and its attributes are set. To finalize the component initialization, the Container will call the *on_init* method on the component body, i.e. the ComponentObject entity.

Then, the component instance is ***configured*** upon successive calls to the *connect* Container method that connect the different component required ports. When all these ports are connected, the component instance state is set to *Configured*. If the component body implements the *Connectable* interface, it is notified on each connection establishment via a call to the *on_connect* method. The component instance may come back to the Initialized state if all its connections are undone upon a call to the *disconnect* method on the container.

Once all the application components instances are properly configured, they become ***ready*** when the *start* method is called on the container. This call signals the end of the configuration phase and the beginning of the operational phase. Hence, each component instance is ready to run and to interact with its environment. The component instance may get shutdown upon a call to the *stop* method on the container. That call moves the instance to the Configured state after having notified the component body of its shutdown.

A component instance may be removed at any time using a call to the remove method of the container. To be removed, a n instance should be shutdown then disconnected first

The Ready state is not the only state of an operational component instance. Typically, its execution policy can make it evolve to other states that are specific to that policy and are handled by the fragments that implement that technical policy.

12. IDL Platform Specific Model for UCM

This section presents the IDL mapping of the UCM meta-model. It provides a set of transformation rules that refine a UCM model into an IDL description. The IDL PSM allows the UCM model to be driven towards its actual implementation. Unlike the equivalent IDL description given earlier, this PSM represents a step towards the component implementation. It is a way to specify the component implementation elements in a programming-language-independent way. And to benefit from the different standard IDL to languages mappings to implement UCM applications.

12.1 Concerned IDL building blocks

It's important to note that even if we rely on the following building blocks (BB), UCM does not (and should not) allow representing their whole expressiveness. This means there are structures that can be defined in IDL with the following building blocks which have no meaning in UCM. That isn't a problem as we are stating a projection from UCM to IDL and not the other way around.

From the IDL separation of the grammar in building blocks, we retained the following

- BB Basic core – Core Data Types
- BB Annotations
- BB Interface – Basic

12.2 General notes on data types mapping

The mapping of data types relies on existing IDL types augmented with UCM-dedicated annotations where needed. The set of defined annotations is specified in Annex 14. These annotations shall be taken into account for IDL compilers to be UCM compliant. Considering anonymous types, every UCM type has an identifier. That facilitates the mapping to IDL in which anonymous types have been deprecated. Thus, several UCM types will be matched on a combination of an IDL typedef and the corresponding type declaration. See section 12.4 for more details.

12.3 Primitive types mapping

12.3.1 Mapping to IDL basic types

The mapping between the UCM built-in types defined in the UCM meta-model and the IDL data types are defined as follows:

UCM primitive type	IDL primitive type
BYTE	octet + @int8
SHORT	short
LONG	long
LONGLONG	long long
UBYTE	octet + @uint8
USHORT	unsigned short
ULONG	unsigned long
ULONGLONG	unsigned long
FLOAT	float
DOUBLE	double
CHAR	char

WCHAR	wchar
BOOLEAN	bool

12.3.2 Annotations for 8 bits integers

For int8 and uint8, there is no support for “signed versus unsigned” 8-bits in IDL. Thus, we need to use octet and add an annotation to tell IDL compilers to map it to the appropriate language type.

12.4 Complex data types mapping

12.4.1 Mapping to IDL constructed types

The mapping between the UCM composite types and the IDL data types is defined as follows:

12.4.1.1 Annotation for native types

A native type can be declared to as long as it provides enough information for performing memory footprint analysis.

```
//IDL
@Annotation MemoryFootprint {
   attribute unsigned long max;
}
```

As an example, a native type of maximum size 1024 bytes should be defined like this:

```
//IDL
@MemoryFootPrint(max=1024)
native MyNativeType;
```

12.4.1.2 Annotation for specifying index types

```
//IDL
@Annotation IndexType {
   attribute String typeid;
}
```

Of course, the type must be resolvable as an integer.

This annotation may be ignored for languages that don’t support such specification.

```
//IDL
@IndexType(typeid=”unsigned long”)
typedef sequence<short, 64> MyShortSequence;
```

```
//IDL
@Range(min=”0”, max=”64”)
@Uint8
typedef octet IdxType;

@IndexType(typeid=”IdxType”)
typedef short[64] MyShortArray;
```

```
//IDL
@IndexType(typeid="short")
enum Color {
  RED,
  GREEN,
  BLUE,
}
```

12.4.1.3 Annotation for specifying default values

```
//IDL
@Annotation DefaultValue {
   attribute String val;
}
```

```
//IDL
@IndexType(typeid="octet")
enum Color {
  @DefaultValue(val="0xFF0000")
  RED,
  @DefaultValue(val="0x00FF00")
  GREEN,
  @DefaultValue(val="0x0000FF")
  BLUE,
}
```

12.5 Constants mapping

A UCM constant is simply translated to an IDL constant.

12.6 Interfaces and exceptions mapping

UCM exceptions can be translated to IDL exceptions as they share the same representation.

UCM interfaces can be translated to IDL interfaces with the same name and the same set of operations. UCM operations map naturally to IDL ones within the IDL interface.

12.7 UCM modules mapping

The UCM meta-model defines specific modules to organize the specification of the components, the contracts, the interactions and the technical policies. All these modules realizes a common abstract meta-class which is *IModule*. All the *IModule*-derived meta-classes, including *ComponentModule*, *ContractModule*, *InteractionDefinitionModule*, *TechnicalAspectModule*, are mapped to IDL modules. Each *IModule*-derived element of a UCM model maps to an IDL module with the same name and including the IDL constructs that map to the *IModule* children elements.

If the UCM *IModule* includes a template parameter definition (*ITempaltedParameter*-derived meta-classes), the corresponding IDL module becomes a template module whose parameter is the IModule template parameter. The IDL template parameter name will be the capitalized name of the IModule one. If this parameter is an *InterfaceTemplateParameter*, the idl parameter will be tagged as an "*interface*". If it is a *DataTypeTemplateParameter*, the idl module parameter will be tagged generically with the "*typename*" keyword. If a UCM *IModule* includes an element that uses a *ITemplatedParameter* defined in a different UCM *IModule*, its equivalent module becomes a template one as well.

Example

In UCM:

```
<InteractionModule name="services">
<ContractModule
    name="contracts">
  <InterfaceTemplateParameter
```

```
        name="service_intf_t"/>
</ContractModule>

<ContractModule name="data">
  <DataTypeTemplateParameter
         name="message_type_t" />
</ContractModule>

<TemplatedPort name="service_server_tport">
  <PortElement name="api"
          intf="contracts:service_intf_t"
          kind="PROVIDED"/>
</TemplatedPort>

</InteractionModule>
```

In IDL:

```
module services<interface SERVICE_INTF_T>
{
 module contracts<interface SERVICE_INTF_T>
 {
 };

 module data<typename MESSAGE_TYPE_T>
 {
 };

...

};
```

12.8 Component Mapping

This section defines the mapping of a UCM *ComponentModule* content including, *ComponentType*, *Port*, *AtomicComponentImplementation* and *ComponentTechnicalPolicy* elements of a UCM model.

The component mapping is used for driving the UCM components implementations. It describes the interfaces that shall be used to implement them.

12.8.1 Component Type mapping

Each ComponentType maps to a single IDL interface called the component equivalent interface. This interface is defined by the following rules:

- Each ComponentType named **<component_name>** maps to an interface having the same name as the component.
- The equivalent interface declares the same set of attributes as the component.
- If the ComponentType has a base ComponentType, its equivalent interface inherits from the base ComponentType's equivalent interface also.
- The ports included in each ComponentType are mapped to a set of IDL operations within the component equivalent interface as presented in section 12.8.4

Example

UCM (in IDL syntax):

```
component TellerComponent {
 readonly attribute short id;
 port  service_server_tport teller;
};
```

IDL mapping:

```
interface TellerComponent {
  readonly attribute short id;
  // ports mapping operations
};
```

12.8.2 Atomic Component Implementation mapping

Each atomic component implementation maps to a single IDL interface, representing the component body interface. This interface provides the component business logic implementation as well as the different callback operations needed by its infrastructure.

The component body interface is defined as follows:

- For an implementation named **<component_impl_name>**, an interface named **<component_impl_name>_Body** is generated.
- The body interface inherits from the component equivalent interface and the ***ComponentObject*** one (§ 11.2.1.2).

The body interface includes a set of additional operations mapped from its related technical policies if any. The details are given in the following sections.

Example:

UCM:

```
<AtomicComponentImplementation language="cpp" name="TellerComponentImpl"
 type="TellerComponentImpl">
  ...
</AtomicComponentImplementation>

<AtomicComponentImplementation language="cpp"
 name="PrinterComponentImpl" type="PrinterComponentImpl">
  ...
</AtomicComponentImplementation>
```

IDL:

```
interface TellerComponentImpl_Body : ComponentObject,
                                     TellerComponent {
 // technical policies mapping operations
};

interface PrinterComponentImpl_Body: ComponentObject,
                                  PrinterComponent {
 // technical policies mapping operations
};
```

12.8.3 Ports elements mapping

Each interface provided or required by a port element is mapped into an IDL interface with the same name and inheriting from the *PortElementObject* interface (§11.2.1.1).

12.8.4 Ports mapping

Component ports refer to either a port role or to a port type as defined by the UCM meta-model. UCM port roles have no IDL mapping, as they do not specify any API. A port type is associated to a *PortAPI* that has been defined as an abstract element in the UCM meta-model. The *TemplatedPort* element has been proposed as a possible way to specify a Port API. The present mapping applies on *TemplatedPort* port types only.

Each Port having a TemplatedPort API is mapped on a set of getter operations making part of the component equivalent interface. The getter operations allows the component to provide its provided ports elements to its infrastructure. These operations are defined as follows:

- For each port named <port_name> having a type with a provided port element named <provided_port_element_name>, a getter operation is generated as part of the component body interface; its name is "*get_<port_name>_<provided_port_element_name>*". This operation has no arguments and returns a reference to the port element actual interface.

Example

Remember that the *service_client_tport* port definition (§ 10.1.5) includes a required port element named 'api', and the *service_server_tport* port specification includes a provided element named 'api' as well. This implies the following IDL mapping:

```
interface HelloInterface : PortElementObject {
};
interface TellerComponent {
 HelloInterface get_teller_api();
};
interface PrinterComponent {
 // no getter operation because of no provided ports elements
};
```

12.8.5 Component Technical policies mapping

Technical policies can specify provided or required interfaces. Provided interfaces must be implemented by the component code and the required ones must be implemented by the infrastructure code. As for the component ports mapping, the component technical policies are mapped to a set of getter operations as part of the component body interface.

For each TechnicalPolicy named <tech_policy> applied on a atomic component implementation and requiring a PortElement named <callback>, a getter operation named "get_tp_<tech_policy>_<callback>" is generated as part of the component body interface.

Example:

Assuming the example components implementation is tied to the predefined self-executing policy and Trace technical policies. Lets recall that the self-executing policy requires an interface from the component, unlike the Trace policy that provides one.

```
<ComponentTechnicalPolicy name="ExecPolicy"
           technicalPolicyDefinition="::policy_mod::protected_self-
executing_component">
...
</ComponentTechnicalPolicy>

<ComponentTechnicalPolicy name="TracePolicy"
           technicalPolicyDefinition="::policy_mod::component_trace">
...
</ComponentTechnicalPolicy>

<AtomicComponentImplementation language="cpp" name="TellerComponentImpl"
                               type="TellerComponentImpl">

  <TechnicalPolicy name="ExecPolicy" />
  <TechnicalPolicy name="TracePolicy" />

</AtomicComponentImplementation>
```

```
<AtomicComponentImplementation language="cpp" name="PrinterComponentImpl"
                               type="PrinterComponentImpl">

  <TechnicalPolicy name="ExecPolicy" />
  <TechnicalPolicy name="TracePolicy" />

</AtomicComponentImplementation>
```

IDL:

```
interface TellerComponentImpl_Body : TellerComponent,
                                 ComponentObject {

  // technical policies related operations
  component_execution_intf get_tp_TellerExecPolicy_self_execution_api();

  // no operation for the Trace policy
};
```

12.9 Interaction definitions Mapping

UCM interactions are specified using the elements included in the ucm_interactions package. These definitions are provided by the platform provider. As stated previously, the UCM standard does not target connectors implementation portability. As a result, there is no IDL mapping for the ***ConnectorDefinition***, ***ConnectorPortDefinition, ConnectorImplementation*** and ***TemplatedPort*** elements.

12.10 Container Programming Model

The container programming model interfaces, as defined in the main UCM document, are defined in IDL as follows:

```
struct Property
{
  string name;
  string value;
};
typedef sequence<Property> Properties;
typedef long componentId;
typedef sequence<componentId> componentIds;
typedef long connectionId;

exception NOT_FOUND{
};
exception BAD_PARAMETER {
};
exception UCM_ERROR {
};

interface PortElementObject {
};
typedef sequence<PortElementObject> PortElementObjects;

struct PortConnection {
  string from_port;
  PortElementObjects to_port;
};
typedef sequence<PortConnection>  PortConnections;

interface ComponentObject {
 void on_init ();
 void on_remove ();
 void on_startup (in PortConnections dependencies);
 void on_shutdown ();
```

```
 PortElementObject get_portElement(in string provided);
};

interface FragmentObject {
  void set_container_interface(in Container container);
};

interface Connectable {
  void on_connect(in string port_name, in PortElementObject required);
  void on_disconnect(in string port_name);
};

interface Container {
  componentId add_component( in Properties configValues) raises (UCM_ERROR);
  void remove_component (in componentId comp_instance) raises (NOT_FOUND, UCM_ERROR);
  PortElementObject get_port_element(in componentId comp_instance, in string port_name) raises
(NOT_FOUND, UCM_ERROR);
  connectionId connect (in componentId instance, in string port_name, in PortElementObject to_port)
raises (NOT_FOUND, BAD_PARAMETER, UCM_ERROR);
  void disconnect (in connectionId connection) raises (NOT_FOUND, UCM_ERROR);
  void start() raises (UCM_ERROR);
  void stop() raises (UCM_ERROR);
  componentIds get_components() raises (UCM_ERROR);
};

interface ContainerManager : Container {
  Container create_container (in Properties configValues) raises (UCM_ERROR);
  void remove_container(in Container subContainer) raises (NOT_FOUND, UCM_ERROR);
  void destroy() raises (UCM_ERROR);
};
```

12.11 Standard Technical Policies Mapping

As stated in the container programming model section, the technical policies implementation is out of scope of the UCM standard. All what should be specified is the contracts between the technical policies implementations and the components, ie. the ports elements of the technical policies. This sections presents the IDL mapping of those contracts. For more information on the technical policies model and semantics, please refer to the UCM meta-model.

12.11.1 Execution Policies

```
module standard_execution_policies {

  module execution_policies_api {

    interface component_execution_intf {
      void run();
    };

  };
};
```

12.11.2 Clock And Trace Services

```
module standard_services {

  module standard_services_api {
    typedef unsigned long ucm_time_t;
```

```
    typedef long  ucm_suseconds_t;
    typedef string<255>  log_message_t;

    struct ucm_timeval_t {
      ucm_time_t utv_sec;
      ucm_suseconds_t utv_usec;
    };

    enum log_severity_t {
     debug,
     info,
     warning,
     error,
    fatal
    };

    interface clock_service_intf : PortElementObject {
      void get_local_time(out ucm_timeval_t local_time);
      void get_synchronized_time(out ucm_timeval_t local_time);
    };

    interface trace_service_intf : PortElementObject {
      void log(in log_severity_t severity, in log_message_t message);
    };
  };
};
```

12.11.3 Advanced Timer Service

```
module timer_service {

  module timer_contracts {

     enum timeout_enum_t {
       ABSOLUTE_TIME,
       RELATIVE_TIME
     };

    struct timeout_t {
      standard_services:ucm_time_t time_val;
      timeou_enum_t flag;
    };

    typedef unsigned long timer_number_t;
    typedef boolean timer_bool_t;

    interface timer_callback_intf : PortElementObject{
      void on_timeout(in timeout_t time, in timer_number_t timer_number);
    };

    interface timer_service_intf : PortElementObject {
      void start_periodic_scheduler(in timer_number_t timer_number, in timeout_t delay_time, in
timeout_t rate);
      void start_sporadic_scheduler(in timer_number_t timer_number, in timeout_t time);
      void cancel_timer(in timer_number_t timer_number);
      timer_bool_t is_canceled(in timer_number_t timer_number);
    };
  };
};
```

12.12 Component Programming Model

Given the IDL mapping rules described above, the component developer shall implement the component body IDL interface as well as its provided ports elements interfaces. Figure 53 depicts this requirement.

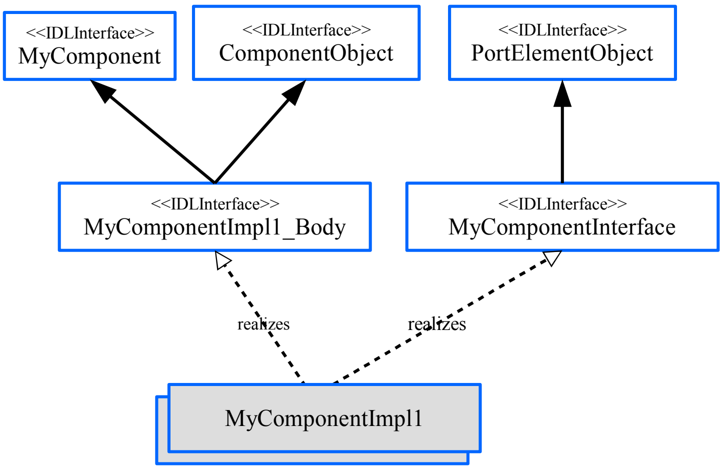

Figure 53: generated IDL interfaces

The component developer may provide one or more programing artifacts to implement the required IDL interfaces. In an object-oriented programing, one or more classes may be used to implement the component body interface and all the provided ports elements ones.

The component body implementation must include:

- the business logic of the different life cycle methods that are defined in the base ComponentObject interface.The on_startup method will provide the component dependencies references that should be stored by the component body for further usage.
- the implementation of the getter operations that should return references to the provided ports elements implementations.
- The component provided ports elements implementations must implement the business methods of the related interface.

All the UCM interfaces must be considered as local interfaces. They describe the interactions between the components and their infrastructure that are necessarily co-localized. Any remote interfaces are managed by some connector fragments and is beyond the scope of this specification.

12.12.1 Middleware-agnostic language mappings

As UCM aims to build middleware-agnostic component frameworks, it is highly recommended to use middleware-agnostic programming-languages mappings for IDL. The class generated from an IDL interface shall not extend any CORBA-specific object such as CORBA::Object or CORBA::LocalObject.

13. C++ Platform Specific Model for UCM

This section presents a native C++ PSM for the UCM metamodel. This PSM is proposed for those who would like to not use IDL as an intermediate step for the components implementation. Given the concepts similarities between the UCM meta-model and the IDL language, this PSM is mainly inspired from the IDL to C++11 standard. This latter rethought the old C++ mapping in order to reduce the dependency to CORBA, to simplify it, and to exploit the modern constructs and capabilities of the latest versions of C++. However, this mapping is still not completely independent from CORBA as it still consider an IDL interface as a CORBA object and IDL exceptions as CORBA exceptions. The proposed PSM lifts this requirement. It reuses most of the mapping rules of that standard except for the interfaces and the exceptions. All UCM interfaces are considered as local C++ objects without any assumed middleware-specific locality meaning. The current PSM is considered as an IDL-independent local C++ PSM derived from the IDL2CPP11 standard. This relationship with the IDL2CPP11 standard is meant to ease the portage of UCM applications from one PSM to the other.

Given the slow adoption of C++11 in the DRTE era, a C++03 PSM is also proposed to not be tied to the specific C++11 features.

13.1 Primitive types mapping

The following table sums up the C++ mapping of all the UCM primitive types.

UCM primitive type	C++ primitive type
BYTE	uint8_t
SHORT	int16_t
LONG	int32_t
LONGLONG	int64_t
UBYTE	uint8_t
USHORT	uint16_t
ULONG	uint32_t
ULONGLONG	uint64_t
FLOAT	float
DOUBLE	double
CHAR	char
WCHAR	wchar_t
BOOLEAN	bool

13.2 Complex data types mapping

The following table sums up the mapping of the different UCM complex types to C++.

UCM composite type	C++ type
Alias	typedef
Sequence	Bounded => std::array Unbounded => std::vector

String8	std::string
String32	std::wstring
Structure	C++ class
Union	C++ class
Enumeration	C++ typed enum
Array	std::array
Constant	const
Native	C++ type

Every data declaration in the UCM model maps to a C++ data declaration whose identifier is the same as the UCM one and the type is derived following the previous mapping table.

13.2.1 Structure mapping

A UCM structure is mapped to a C++ class as defined by the IDL2CPP11 specification.

13.2.2 Union mapping

A UCM union is mapped to a C++ class as defined by the IDL2CPP11 specification.

13.2.3 Enumeration mapping

A UCM enumeration maps to a C++11 enum as defined by the IDL2CPP11 specification.

An example is given below. In UCM:

```
<Enumeration name="Shape" type="@ushort">
 <Enumerator name="triangle" value="10"/>
 <Enumerator name="square" value="20"/>
 <Enumerator name="circle" value="30"/>
</Enumeration>
```

In C++:

```
enum Shape : unit8_t {
    triangle = 10,
    square = 20,
    circle = 30
  };
};
```

13.2.4 Array mapping

A UCM array maps to the standard std::array<> type as defined in the IDL2CPP11 specification. The array dimension maps naturally to the std::array size. The index type has no equivalent construct in C++.

13.2.5 Sequence mapping

A UCM sequence may be bounded if its max size is set to a positive non null integer, or unbounded if its max size is not set or is set to a negative value. A bounded sequence maps to the C++ std::array<> type as it is a fixed size collection of elements. An unbounded sequence maps to the C++ std::vector<> type that is a dynamic size collection of elements. The index type has no equivalent construct in C++.

13.2.6 String mapping

A UCM String8 sequence maps to the C++ std::string type. And the UCM String32 type maps to std::wstring.

13.2.7 Constant mapping

A UCM constant maps to a C++ constant.

13.3 UCM Module mapping

A UCM IModule element maps to a C++ namespace with the same name. This namespace will include all the C++ definitions corresponding to the UCM elements included in the IModule.

13.4 Exception Mapping

A UCM exception maps to a C++ class following the same rules as the IDL2CPP11 standard, but without the CORBA-specific concepts.

- All the UCM exceptions implement a common abstract class UCM::Exception that is similar to the Exception class defined in IDL2CPP11, but without the rep_id() method. This method returns the repository id of the exception which is a CORBA-specific concept.
- User exceptions does not inherit from CORBA::UserException. Instead, a UCM::UserException is defined as a root to all user exceptions.
- System exception does not inherit from CORBA::SystemException. Instead, UCM::SystemException is defined as a root to all runtime exceptions.

13.5 Attribute Mapping

Whether the attribute belongs to a UCM interface or a component type, its C++ mapping is the same. Each read-write attribute maps to a pair of public pure virtual C++ functions having the same name as the UCM attribute. One accessor function that returns the attribute value, and one mutator function that sets the attribute value. A read-only attribute will map to an accessor function only. In UCM:

```
<Interface name="Logger" >
 <Attribute name="name" type="string8" mode ="READ"/>
 <Attribute name="level" type="LogLevelsEnum"/>
</Interface>
```

In C++:

```
// LogLevelsEnum enumeration mapping

class Abstract_Logger {

public:
  virtual std::string name() = 0;

  virtual LogLevelsEnum level() = 0;
  virtual void level(LogLevelsEnum l) = 0;

};
```

13.6 Interface Mapping

A UCM interface maps to a C++ abstract class to translate the general concepts that the UCM interface defines. It will typically be used as a base class for concrete implementation classes. The C++ abstract class is named following the pattern “Abstract_<interface name>”. It includes the C++ mapping of the attributes and the operations defined within the UCM interface. If the UCM interface inherits from other interfaces, its equivalent C++ class will also inherit from the equivalent C++ classes of the base interfaces, using a public inheritance.

13.6.1 Operations Mapping

Each operation maps to a pure virtual function with the same name and the same set of parameters. The parameter passing modes depend on their types and direction. All out and inout parameters are passed by reference whatever their types. Primitive types defined as IN parameters are passed by value. The other types are rather passed as const references. This is similar to the IDL2CPP11 specification.

UCM parameter direction of Primitive types	C++ parameter passing
IN T	T
OUT T	T &
INOUT T	T &
RETURN T	T

UCM parameter direction of non primitive types	C++ parameter passing
IN T	const T &
OUT T	T &
INOUT T	T &
RETURN T	T

13.6.2 Interface Reference Mapping

A reference to a UCM interface within a UCM model maps to a C++ shared pointer (std::shared_ptr) to its related class. A recurring theme for C++ programmers is the need to deal with memory allocations and deallocations in their programs. It can be extremely difficult to ensure that a program does not leak resources, if ownership of dynamic memory is not properly tracked. C++ shared pointers usage allows this problem to be resolved. It uses reference counting to keep track of each class instance and, when the last reference disappears, automatically delete the instance. Hence, when a UCM interface T is passed as a parameter of a given operation, its C++ mapping is given in the following table:

UCM parameter direction of interface type	C++ parameter passing
IN T	std::shared<T>
OUT T	std::shared<T> &
INOUT T	std::shared<T> &
RETURN T	std::shared<T>

A shared pointer is created using std::make_shared.

13.7 Component Mapping

Both the UCM AtomicComponentImplementation and its related ComponentType map on a common C++ abstract class. This class represents the interface that should be implemented by the component body. It includes a set of pure virtual methods corresponding to the component attributes, ports and technical policies. Besides all the methods needed to manage the component life cycle and to enable its operation at runtime.

- The class name is the same as the related atomic component implementation, prefixed by “Abstract_”
- It includes four life cycle management methods
  - void on_startup(const MyPort_Connection & p1, ...)
  - void on_init()
  - void on_shutdown()
  - void on_remove()

The on_startup method provides the component dependencies references to the component body. For each port named <port_name> having a port type with a required port element named <required_port_element_name>, a new type named “<port_name>_<port_element_name>_Connection” is defined from the PortConnection template structure. This latter is a struct template that associates a string that denotes the port name concatenated to the port element name, with a reference to the actual type of the port element. The on_startup method will have as much parameters as required ports elements connection objects. These objects will typically be used to invoke the required port elements interfaces.

For a component having a port named “Filter_in” with a required port element named “emitter_port_element” of type “Message_Intf”, the on_startup method is generated as follows:

```
typedef                                        PortConnection<Message_Intf>
Filter_in_emitter_port_element_Connection;

void on_startup (const Filter_in_emitter_port_element_Connection &cnx);
```

knowing that PortConnection is defined as follows:

```
template<class T>
struct PortConnection
{
  std::string port_name;
  std::shared_ptr<T> ref;

  PortConnection(std::string name, std::shared_ptr<T> ref): port_name(name),
                                                            ref(ref)
  {}

};
```

- It includes the methods corresponding to the related component type attributes if any (section 13.5).
- For each port, within the component type, named <port_name> having a port type with a provided port element named <provided_port_element_name>, a public getter method is generated as part of the component body interface. Its name is “get_<port_name>_<provided_port_element_name>”. This operation has no arguments and returns a reference to the port element actual interface. It allows the component body to provide a reference to the implementation of that interface.
- For each technical policy, within the component atomic implementation, named <tech_policy> with a required port element named <callback>, a getter operation named “get_tp_<tech_policy>_<callback>” is generated as part of the component body interface. This method returns a reference to the port element actual interface. This method allows the component to provide its implementation of the required interface to the platform.

In UCM:

```
<ComponentType name="Filter">
  <Port name="Filter_in">
    <PortTypeSpec type="Messages::message_receiver_port">
```

```
        <AbstractTypeBinding abstractType="Messages::message_type_t"
          actualType="coordinate_t"/>
    </PortTypeSpec>
  </Port>
  <Port name="Filter_out">
     <PortTypeSpec type="Messages::message_emitter_port">
       <AbstractTypeBinding abstractType="message_type_t"
          actualType="coordinate_t"/>
    </PortTypeSpec>
   </Port>
</ComponentType>
<AtomicComponentImplementation language="cpp" name="Filter_Impl" type="Filter">
...
</AtomicComponentImplementation>
```

In C++:

```
typedef PortConnection<Message_Intf> Filter_in_emitter_port_element_Connection;


class Filter_impl_Abstract {

public:

//life cycle methods
virtual void on_init() = 0 ;
virtual void on_startup (const Filter_in_emitter_port_element_Connection & cnx1)
= 0;
virtual void on_shutdown() = 0;
virtual void on_remove() = 0;

// provided port elements getters
Message_Intf get_Filter_out_receiver_port_element() = 0;

}

...
```

13.8 Ports elements interfaces mapping

Each UCM interface referenced by a port element maps to a C++ abstract class, as defined in section 13.6, that must inherit from PortElementObject (§ 11.2.1.1). This latter has no operations. It is used for characterizing ports elements interfaces implementations.

13.9 Component Programming Model

There are two strategies for the implementation of the component body:

- Typed component body class

  In this case, the user will implement the abstract component body class as described in section 13.7. This class is tailored for specific components types as it appears from its methods signatures. The interface between the component implementation and its framework is component-type-dependant. It is heavily based on code generation. All or part of the deployment code is generated to deal with specific component implementations. This approach allows to have statically-typed code with less type casting and less dead code. This is because the component framework manages the component implementations using their specific types. This approach leads to faster, tighter and safer applications but increases the cost of change. Any change in the component type may lead to a complete generation, re-compilation and re-qualification of the application.

- Generic component body class

  In this case, the component body will implement the pre-defined ComponentObject (§ 11.2.1.2) interface. This interface includes semantically-equivalent methods to the typed interface. The only differences are in the methods signatures.

The platform provider may enable both or one of the two strategies for components implementation.

Whatever the chosen component implementation strategy, the developer must provide the following implementations:

- A concrete implementation for the component body abstract class (§ 13.7)
- Concrete implementations for the components provided ports elements abstract classes.

13.10 Derived C++03 PSM

Given the slow adoption of the C++11 language, this section defines an ISO C++ 2003 for UCM. It is derived from the previously described C++11 PSM by substituting all the C++11 specific features by C++03 ones.

All the mapping rules stated previously for the C++11 language remains valid but with the constrains presented in the following table:

C++11 feature	C++03 feature
nullptr	NULL or 0
std::shared_ptr	ucm::shared_ptr
strongly typed enum	ucm::safe_enum
constexpr	“const”, but floats and double cannot be defined in the header file
std::array	ucm::array
R-value references	Not available. No move semantics.
user defined literals	Not supported
final/override	Not supported

Some of the C++11 specific features could be replaced by others in C++03 like “nullptr” and “constexpr”. C++11 shared pointers, typed enumerations and arrays may be implemented by the UCM framework itself. The remaining features cannot be supported and will then not be available in the C++03 PSM.

13.10.1 Array mapping

A UCM array maps to a C++ class or struct in the ucm namespace that provides std::array semantics.

13.10.2 Enumeration mapping

This PSM maps the UCM Enumeration to a C++ class which is similar to the DDS Enumeration mapping (formal/13-11-01).

```
namespace ucm
{
   template<typename def, typename inner = typename def::type>
   class safe_enum : public def {
```

```
    typedef typename def::type type;
    inner val;

    public:
     safe_enum(type v) : val(v)
     {}

     inner underlying() const {
       return val;
     }

    bool operator == (const safe_enum & s) const;
    bool operator != (const safe_enum & s) const;
    bool operator < (const safe_enum & s) const;
    bool operator <= (const safe_enum & s) const;
    bool operator > (const safe_enum & s) const;
    bool operator >= (const safe_enum & s) const;

  };
}
```

Hence, a UCM enumeration maps to:

- a C++ struct named following the pattern “<enum_name>_def”, and including an enum declaration named “type” and having the same enumerators names as the corresponding UCM enumeration.
- a ucm::safe_enum class instance type named as the UCM enumeration name and instanciated with two parameters: the previous C++ struct and the type of the underlying enumerators.

An example is given below:

In UCM:

```
<Enumeration name=”Shape” type=”ushort” >
 <Enumerator name=”triangle” value=”10”/>
 <Enumerator name=”square” value=”20”/>
 <Enumerator name=”circle” value=”30”/>
</Enumeration>
```

In C++:

```
struct Shape_def {
 enum type {
    triangle = 10,
    square = 20,
    circle = 30
  };
};
typedef ucm::safe_enum<Shape_def, uin8_t>
        Shape;

// declaring a triangle shape
Shape s = Shape::triangle;
```

13.10.3 Interface reference mapping

Interface references in a UCM model maps to a C++ template class ucm::shared_ptr that provides the std::shared_ptr semantics.

14. Summary of UCM IDL annotations

```
//List of annotations added to enrich information on datatypes.

@Annotation int8 {
}

@Annotation uint8 {
}

@Annotation MemoryFootprint {
   attribute unsigned long max;
}

@Annotation IndexType {
   attribute String typeid;
}

@Annotation DefaultValue {
   attribute String val;
}
```